



저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:



저작자표시. 귀하는 원저작자를 표시하여야 합니다.



비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 [이용허락규약\(Legal Code\)](#)을 이해하기 쉽게 요약한 것입니다.

[Disclaimer](#)

공학박사 학위논문

Architecture Design and Compiler Support for Code Size Optimization in Embedded Processors

내장형 프로세서에서의 코드 크기 최적화를 위한 아키텍처
설계 및 컴파일러 지원

2014년 2월

서울대학교 대학원
전기 컴퓨터 공학부
이 종 원

공학박사 학위논문

Architecture Design and Compiler Support for Code Size Optimization in Embedded Processors

내장형 프로세서에서의 코드 크기 최적화를 위한 아키텍처
설계 및 컴파일러 지원

2014년 2월

서울대학교 대학원
전기 컴퓨터 공학부
이 종 원

Architecture Design and Compiler Support for Code Size
Optimization in Embedded Processors

내장형 프로세서에서의 코드 크기 최적화를 위한
아키텍처 설계 및 컴파일러 지원

지도교수 백 윤 흥

이 논문을 공학박사 학위논문으로 제출함

2014 년 2 월

서울대학교 대학원

전기 컴퓨터 공학부

이 종 원

이 종 원의 공학박사 학위논문을 인준함

2014 년 2 월

위 원 장	최기영
부위원장	백윤흥
위 원	문수묵
위 원	이종은
위 원	윤종희

Abstract

Embedded processors usually need to satisfy very tight design constraints to achieve low power consumption, small chip area, and high performance. One of the obstacles to meeting these requirements is related to delivering instructions from instruction memory/caches. The size of instruction memory/cache considerably contributes total chip area. Further, frequent access to caches incurs high power/energy consumption and significantly hampers overall system performance due to cache misses. To reduce the negative effects of the instruction delivery, therefore, this study focuses on the sizing of instruction memory/cache through *code size optimization*.

One observation for code size optimization is that very long instruction word (VLIW) architectures often consume more power and memory space than necessary due to long instruction bit-width. One way to lessen this problem is to adopt a *reduced bit-width* ISA (Instruction Set Architecture) that has a narrower instruction word length. In practice, however, it is impossible to convert a given ISA fully into an equivalent reduced bit-width one because the narrow instruction word, due to bit-width restrictions, can encode only a small subset of normal instructions in the original ISA. To explore the possibility of complete conversion of an existing 32-bit ISA into a 16-bit one that supports effectively all 32-bit instructions, we propose the reduced bit-width (e.g. 16-bit \times 4-way) VLIW architectures that equivalently behave as their original bit-width (e.g. 32-bit \times 4-way) architectures with the help of *dynamic implied addressing mode (DIAM)*.

Second, we observe that code duplication techniques have been proposed to increase the reliability against soft errors in multi-issue embedded systems such as VLIW by exploiting empty slots for duplicated instructions. Unfortunately, all duplicated in-

structions cannot be allocated to empty slots, which enforces generating additional VLIW packets to include the duplicated instructions. The increase of code size due to the extra VLIW packets is necessarily accompanied with the enhanced reliability. In order to minimize code size, we propose a novel approach *compiler-assisted dynamic code duplication scheme*, which accepts an assembly code composed of only original instructions as input, and generates duplicated instructions at runtime with the help of encoded information attached to original instructions. Since the duplicates of original instructions are not explicitly present in the assembly code, the increase of code size due to the duplicated instructions can be avoided in the proposed scheme.

Lastly, the third observation is that, to cope with soft errors similarly to the second observation, a recently proposed software-based technique with TMR (Triple Modular Redundancy) implemented on *coarse-grained reconfigurable architectures (CGRA)* incurs the increase of configuration size, which is corresponding to the code size of CGRA, and thus extreme overheads in terms of runtime and energy consumption mainly due to expensive voting mechanisms for the outputs from the triplication of every operation. To reduce the expensive performance overhead due to the large configuration from the validation mechanism, we propose selective validation mechanisms for efficient modular redundancy techniques in the datapath on CGRA. The proposed techniques selectively validate the results at synchronous operations rather than every operation.

Keywords: embedded processor, code size, VLIW architecture, reduced bit-width ISA, DIAM, soft errors, instruction duplication, CGRA, TMR, selective validation

Student Number: 2007-21051

Contents

Abstract	i
Chapter 1 Introduction	1
1.1 Instruction Delivery	1
1.2 The causes of code size increase	2
1.2.1 Instruction Bit-width in VLIW Architectures	2
1.2.2 Instruction Redundancy	3
Chapter 2 Reducing Instruction Bit-width with Dynamic Implied Addressing Mode (DIAM)	7
2.1 Conceptual View	10
2.2 Architecture Design	12
2.2.1 ISA Design	13
2.2.2 Remote Operand Array Buffer	15
2.2.3 Microarchitecture	17
2.3 Compiler Support	22
2.3.1 16-bit Instruction Generation	24
2.3.2 DDG Construction & Scheduling	26

2.4	VLES(Variable Length Execution Set) Architecture with a Reduced Bit-width Instruction Set	29
2.4.1	Architecture Design	30
2.4.2	Compiler Support	34
2.5	Experiments	36
2.5.1	Setup	36
2.5.2	Results	39
2.5.3	Sensitivity Analysis	48
2.6	Related Work	49
Chapter 3	Compiler-assisted Dynamic Code Duplication Scheme for Soft Error Resilient VLIW Architectures	53
3.1	Related Work	54
3.2	Compiler-assisted Dynamic Code Duplication	58
3.2.1	ISA Design	60
3.2.2	Modified Fetch Stage	62
3.3	Compilation Techniques	66
3.3.1	Static Code Duplication Algorithm	67
3.3.2	Vulnerability-aware Duplication Algorithm	68
3.4	Experiments	71
3.4.1	Experimental Setup	71
3.4.2	Effectiveness of Compiler-assisted Dynamic Code Duplication	73
3.4.3	Effectiveness of Vulnerability-aware Duplication Algorithm .	77
Chapter 4	Selective Validation Techniques for Robust CGRAs against Soft Errors	85
4.1	Related Works	86
4.2	Motivation	88

4.3	Our Approach	91
4.3.1	Selective Validation Mechanism	91
4.3.2	Compilation Flow and Performance Analysis	92
4.3.3	Fault Coverage Analysis	96
4.3.4	Our Optimization - Minimizing <i>Store</i> Operation	97
4.4	Experiments	99
4.4.1	Setup	99
4.4.2	Experimental Results	100
Chapter 5	Conculsion	110
	초록	122

List of Figures

Figure 2.1	32-bit ISA vs. reduced ISA	10
Figure 2.2	Conceptual View	11
Figure 2.3	Example of Partial and Complete Instructions	11
Figure 2.4	ISA Conversion	14
Figure 2.5	ROA buffer operations	18
Figure 2.6	Decode Stage with ROA buffer	20
Figure 2.7	Read/Write buffer pointer management	21
Figure 2.8	An example of code generation	23
Figure 2.9	Algorithm for a multi-issue code generation	25
Figure 2.10	Compilation Framework	26
Figure 2.11	An example of 16-bit Instruction generation	27
Figure 2.12	Data Dependence Graph	28
Figure 2.13	Generated Multiple-issue Code with ROA slot	29
Figure 2.14	A prefix instruction for our VLES architecture	31
Figure 2.15	A fetch logic for our VLES architecture	32

Figure 2.16	An example for behavior of our VLES architecture (<i>Pt</i> represents a fetch packet at cycle t ($t=0,1,2,\dots$), <i>FE/DC</i> denotes the pipeline register between fetch(<i>FE</i>) and decode(<i>DC</i>) stages, <i>DC/EX</i> denotes the pipeline register between decode(<i>DC</i>) and execution(<i>EX</i>) stages.)	33
Figure 2.17	Code transformation algorithm for our VLES architecture . . .	35
Figure 2.18	An example of code transformation for our VLES architecture	36
Figure 2.19	Code size of each processor (normalized to 32VLIWP) . . .	42
Figure 2.20	Utilization of NOP slots for ROA slots	42
Figure 2.21	Fetch energy consumption and fetch count of each processor (normalized to 32VLIWP)	43
Figure 2.22	Total energy consumption of each processor (normalized to 32VLIWP)	44
Figure 2.23	Execution time of each processor (normalized to 32VLIWP)	44
Figure 2.24	Energy-delay product of each processor (normalized to 32VLIWP)	45
Figure 2.25	Results of converting 32-bit ISA to 16-bit ISA for each n -way VLIW processor ($n = 2, 3, 4, 5$)	47
Figure 3.1	Our VLIW architecture for Dynamic Code Duplication . . .	59
Figure 3.2	Three possible cases for generating instruction duplication (A shade one represents a duplicate instruction, (D0,D1)=(0,0) means no duplication.)	61
Figure 3.3	Behavior flow chart of modified fetch stage	63
Figure 3.4	An example: Behaviors of modified fetch stage according to each case	65
Figure 3.5	TPVADuplication– Temporal and Physical Vulnerability-aware Duplication Algorithm	70

Figure 3.6	Our Compiler-Simulator-Synthesizer Framework	72
Figure 3.7	The effectiveness in code size reduction	75
Figure 3.8	The effective code size reduction over benchmarks	75
Figure 3.9	The competitiveness in performance	76
Figure 3.10	Effectiveness of our Dynamic Code Duplication with TVAD in terms of Vulnerability and Error Detection	78
Figure 3.11	Effectiveness of Our Dynamic Code Duplication Scheme with PVAD in terms of Vulnerability and Error Detection	80
Figure 3.12	Effectiveness of Our Dynamic Code Duplication with TP- VAD in terms of Vulnerability and Error Detection	81
Figure 3.13	The effectiveness of our compiler-assisted dynamic code du- plication as compared to hardware-based approach	83
Figure 4.1	Our 4×4 CGRA architecture	88
Figure 4.2	A simple kernel and its generated DFGs of base (no redun- dancy), software implemented TMR with the full voting [1], and TMR with the selective voting (our approach)	89
Figure 4.3	Runtime overhead of voting overhead takes up about 64.8% in software implemented TMR techniques on CGRAs.	91
Figure 4.4	Compilation flow for a system with CGRA	92
Figure 4.5	Mapping operations for software implemented TMR onto CGRAs	93
Figure 4.6	Fault coverage analysis of software implemented TMRs in case of double soft errors (faded nodes and edges indicate no more execution in case of uncorrectable error detection at validations in Figure 4.6(c) and Figure 4.6(d))	95
Figure 4.7	Generated DFGs of our optimization technique with the loop unrolling	98

Figure 4.8	Our framework for simulations	99
Figure 4.9	Our selective voting for TMR outperforms the full voting in terms of runtime and energy consumption.	102
Figure 4.10	Our selective comparison for DMR outperforms the full comparison in terms of runtime and energy consumption.	103
Figure 4.11	Our optimization techniques for TMR can improve the performance in terms of runtime and the energy consumption.	106
Figure 4.12	Our optimization techniques for DMR can improve the performance in terms of runtime and the energy consumption.	107
Figure 4.13	Evaluations of runtime and energy consumption among various protection techniques (benchmark: <i>Cvtcolor</i>)	109

List of Tables

Table 2.1	Four processors used in the experiments	38
Table 2.2	Cell area of each processor (unit : μm^2)	40
Table 2.3	Clock cycle time of each processor (unit : ns)	40
Table 3.1	The meaning of D0 and D1 to duplicate instructions dynamically	62
Table 3.2	Cell areas of instructions normalized to that of an instruction <i>ldc_ri</i>	69
Table 3.3	Cell Area Comparisons (unit: μm^2)	73
Table 4.1	CGRA Power Parameters	100

Chapter 1

Introduction

1.1 Instruction Delivery

Today's design requirements of embedded processors demand system architects to consider strict constraints such as high performance, small area, and low energy consumption all simultaneously. One of the obstacles to meeting these requirements is related to delivering instructions from instruction memory/caches. This is because instruction fetch logic consumes a significant portion of total processor energy dissipation; for example, instruction delivery consumes up to 30% [2] of total energy due to the use of instruction caches in embedded processors; and the size of instruction memory/cache considerably contributes total chip area. Further, frequent access to caches incurs high power/energy consumption and significantly hampers overall system performance due to cache misses. Therefore, this study focuses on the sizing of instruction memory/cache through *code size optimization* to reduce the negative effects of the instruction delivery.

1.2 The causes of code size increase

1.2.1 Instruction Bit-width in VLIW Architectures

As multimedia applications such as videos, sounds and images are becoming ever more complex and diverse, embedded systems targeting such applications have an increased tendency to attain a desirable performance with VLIW (very long instruction word) to take advantage of instruction-level parallelism (ILP). VLIW processors are designed to enhance the performance by executing operations in parallel based on a fixed schedule determined by the compiler. Some high-end DSP and media processors attempt to satisfy performance requirements by means of an eight (or more)-way VLIW architecture, which may issue and execute eight or more instructions simultaneously [3]. Unfortunately, as a VLIW processor lengthens its instruction word to encompass more issue slots, the code efficiency tends to be sharply lower. To make the matter worse, widening the word length is also likely to proportionally increase power consumption as well as chip area due to the widened instruction bus, which deeply influences the design of internal processor components such as memory buses and instruction caches. In fact, a recent study [4] reveals that a significant portion of power is consumed by instruction cache, implying that the detrimental effect of widened data path on power can be devastating in reality. This should be a critical setback for low-cost embedded processors, which are subject to tight constraints on code size and power.

One simple way to alleviate this problem ought to be limiting the number of issue slots in a VLIW packet, just in the case of low-cost VLIW processors [5] which usually have four or less issue slots. As an alternative way, the instruction bus width can be reduced by cutting down on the size of each slot. Existing VLIW processors normally adopt a 32-bit instruction set architecture (ISA) like most modern microprocessors. Therefore, for a 4-way VLIW processor the instruction bus must be 128 bits wide to efficiently support the execution of each VLIW packet. However, the width would

be halved into 64 bits if we could convert the original 32-bit ISA into the equivalent 16-bit one for this processor. As stated earlier, the narrower instruction bus design will offer a more efficient hardware implementation in terms of area and power by decreasing bus-bandwidth requirements and reducing the power dissipation associated with instruction fetches [6]. On top of this advantage, by converting a 32-bit ISA into a 16-bit ISA the code size can be reduced substantially (up to 50% in ideal cases) [7, 8]. Of course, all such gain from the conversion does not come at no cost; severe restrictions on the 16-bit ISA make the conversion difficult or sometimes impossible. A major restriction is that the 16-bit ISA suffers from extremely limited encoding space, thus not all instructions from the 32-bit ISA can be converted into 16-bit long ones. Because in part of this, existing microprocessors [9, 10] employ the multiple ISAs where *reduced bit-width* (i.e. 8-bit or 16-bit) instructions are supplemented only as a subset of the normal 32-bit instructions. In their designs, the encoding space restriction usually forces narrower instructions to access only a restricted set of registers. This possibly increases register spills or extra move instructions in code generation, which in turn usually increases the overall code size.

1.2.2 Instruction Redundancy

Instruction Duplication in VLIW Architectures

Several constraints such as performance, code size, area, and power have been posed in designing embedded systems. Besides these constraints, reliability is becoming an important concern for the design of embedded systems [11]. This is because technology scaling, which incurs shrunk feature size, decreased voltage level and reduced noise margin, makes systems more susceptible to transient faults [12, 13, 14]. Transient faults, also known as *soft errors*, mainly caused by energetic particles such as alpha particles and neutrons, may result in erroneous program states, incorrect outputs

and eventually system crashes. Unless soft errors are detected, even though they are not permanent and non-destructive, the reliability of a system cannot be ensured any longer. Especially, in resource-constrained embedded systems used for medical, financial and security applications, requiring reliable information, it is extremely important to deliver high reliability by detecting soft errors with least overheads in terms of code size, area, performance, and power [15, 16].

VLIW (Very Long Instruction Word) architectures are popular in embedded systems since they offer the potential for high-performance processing at a relatively low cost and energy usage [17]. Consequently, techniques to improve the reliability of application execution in VLIW processors are of interest [18]. Several approaches have been proposed to detect soft errors in VLIW architectures. One of promising techniques is to duplicate instructions at compile-time. The idea of duplicating instructions exploits available resources in VLIW architectures. The lack of instruction level parallelism in applications unavoidably makes an amount of issue slots unused. Indeed, a number of slots are unused on the average in 4-way VLIW processors [11]. These unused issue slots are called *empty slots*. By allocating the duplicated instructions to these empty slots possibly at compile-time, comparing the result of an original instruction and that of its duplicate at run-time, and flagging error detection if they are not identical, the reliability of VLIW architectures can be improved [19, 20, 18, 11].

Unfortunately, all duplicated instructions cannot be allocated to empty slots, which enforces generating additional VLIW packets to include the duplicated instructions. Thus, the increase of code size, another important design concern in embedded systems, due to the extra VLIW packets, is necessarily accompanied with the enhanced reliability. For example, Jie Hu et al. [18, 11] have recently proposed a constraint-induced instruction duplication technique for VLIW architectures. However, their technique can increase the code size by up to 90% for complete instruction duplication. The increase of code size has a negative impact not only on the design constraint but also

on the system reliability. This is because a large size of code makes more bits present in the system, which leads to a higher soft error rate since the larger exposed, the more vulnerable [21]. Further, their duplication algorithm does not consider different degrees of vulnerability in instructions and thus it might lose the effectiveness of duplicating instructions by unnecessarily duplicating unimportant instructions in terms of reliability. The main reason is because they try to duplicate instructions in a sequential manner without any awareness of instruction vulnerability. Thus, their approach first duplicates early-located instructions of the code especially when the performance or power constraint is relatively small.

Validation Mechanism on CGRA

Coarse-Grained Reconfigurable Architecture or CGRA is receiving lots of attentions. It is necessary to achieve not only high performance but also power efficiency in recent embedded systems. CGRA is in general composed of grid-based PEs (Processing Elements) and each PE consists of a FU (Functional Unit) and a few registers. CGRA is a promising alternative as an accelerator since this simple architecture can improve the performance massively by executing application loop kernels on PEs in parallel with the inherent efficacy of power consumption. Further, CGRA is programmable, i.e., able to reconfigure architectures by switching CGRA configuration for a new application in the short amount of time. Thus, CGRAs have been used to accelerate complex applications where high performance is required with the power efficiency [22, 23].

Soft error and its concern are on significant increase in embedded system designs. Several decades of technology scaling has brought us where transistors are extremely susceptible to even small fluctuations in supply voltage levels, slight noise in the power, signal interference, and even induced radiation [12, 13, 14]. Any of these effects can temporarily toggle the logic value of a transistor, so it is called a transient fault or soft error. Such a soft error is not permanent and non-destructive, i.e., resetting the

device can resume the normal operation. However, a single soft error can be as critical as a permanent error. Indeed, soft errors have been already revealed to cause significant fiscal damages [24, 25, 26]. As the popularity of CGRA usages is increasing on many embedded applications such as human health systems, automobiles, airplanes, and data server systems [27], a single soft error may lead to catastrophic consequence, and even a human life.

To make CGRAs robust against soft errors, several hardware based techniques have been proposed [27, 28, 29, 30], but they are expensive in terms of area, energy, and performance. Most of hardware based techniques modify existing architectures to implement redundancy based DMR (Dual Modular Redundancy) [31] and TMR [32] and they incur high costs in every design aspect. To resolve these drawbacks from hardware based techniques, researchers move attention to software based techniques that are of no area overhead [33, 1]. Recently, an interesting software based technique has been proposed but it still incurs high performance overhead mainly due to expensive voting and comparison mechanisms of TMR and DMR, respectively [1]. In fact, Lee et al. [1] has demonstrated that software implemented TMR and DMR on 6×8 CGRAs incur up to 700% and 167% performance overheads, respectively.

Chapter 2

Reducing Instruction Bit-width with Dynamic Implied Addressing Mode (DIAM)

The objective of this chapter is to explore the possibility of full conversion of an existing 32-bit VLIW ISA into an equivalent 16-bit one. In other words, main goal is to implement low-end processor with a reduced bit instruction word path. In the experiment, it is given a 4-way VLIW processor whose instruction packet is therefore 128 bits long. Consequently, the converted processor requires only 64 bits for its packet. Virtually all 32-bit instructions in the original processor were successfully mapped to 16-bit ones in the new processor. Also, 16-bit instructions have no restriction on accessing registers unlike those in the processors with dual ISAs. As a result, the compiler for the original processor was able to transform the 32-bit assembly code into the code for the 16-bit ISA via a simple translation based on these mapping rules.

The key idea behind this work is dynamic insertion of *excessive* operands from the original 32-bit instructions into the 16-bit instructions. By excessive, we mean some

operands in a 32-bit instruction that cannot be fit into its 16-bit counterpart. As an example, consider an ALU instruction, `xor r1, r2, r3`, in the original processor. In the 32-bit ISA design, the instruction takes up 22 bits in total, leaving 10 unused bits¹. As the original processor has about 100 or so instructions, we provide 7 bits for the opcode field within the 16-bit issue slot. Then we have only 9 bits left for encoding the operands. However, since the processor must designate 16 registers in its register operand field, we need at least 12 bits for the three operands, `r1`, `r2` and `r3`. Obviously in this case, encoding all three operands goes beyond the 16-bit length limit. Therefore, only two operands, say `r1` and `r2`, can be encoded inside the 16-bit issue slot, and the remaining one `r3` becomes excessive. We hereby say that this 16-bit instruction encoded with the opcode and two operands is a *partial instruction*, and the excessive register operand is a *remote operand*. Notice that this partial instruction and its remote operand must be coupled to behave like the original instruction. So in our work, at compile time, all remote operands are relocated and stored to some other issue slots close to their *coupled* partial instructions. Then at run time, remote operands are dynamically retrieved and inserted into proper partial instructions by hardware to form complete instructions for execution. For this, we propose a VLIW compiler that can generate and schedule partial instructions as well as remote operands, and also a VLIW pipeline architecture that can construct 32-bit instructions dynamically from its 16-bit issue slots during its decode stage.

The question now is where remote operands are relocated and stored at compile time. Indiscriminate relocation of these operands may demand many new slots created for them, hence increasing the overall size of VLIW code. A possible solution for this question can be derived from the observation that in VLIW code generation, a fair number of issues slots are unused and filled with *NOPs* (No Operations). In fact, *code*

¹32-bit instructions of existing processors like ARM usually contain unused or don't-care bits, which are rare in 16-bit instructions. Although these bits may be reserved for future use, we believe that they are one reason that causes code size to increase unnecessarily.

bloating due to many NOPs has been a serious problem in VLIW compilers [34], and various approaches have been suggested to tackle this problem [35, 36, 37]. However, according to empirical studies [38], despite all these efforts, on a 4-way VLIW processor about one slot per VLIW packet on average is still wasted to run NOPs. In our approach, we strive to store as many remote operands as possible into these wasted slots, thereby deterring the increase of code size due to the allocation of remote operands.

In addition to the 32-bit ISA VLIW processor, we also apply the ISA conversion to an existing VLIW processor with a VLES (variable length execution set) architecture, called a *VLES processor* in this dissertation. The VLES architecture is successfully adopted in modern VLIW processors [3] [39] since it can considerably resolve the code bloating problem due to NOPs by removing them in the code. To make our approach more practical in modern VLIW-based embedded systems, we also propose the concept of architectural design and compiler code generation algorithm for the VLES architecture.

The rest of the chapter is organized as follows. we will explain in more detail the basic strategy of our approach in Section 2.1. Then, we will introduce the proposed VLIW pipeline architecture in Section 2.2. The compiler support is indispensable for generating an efficient code that fully utilizes the suggested approach. In Section 2.3, we introduce a new scheduling algorithm that performs static program analysis to help the compiler to translate the 32-bit assembly code into our new VLIW processor. In Section 2.4, we will show how the ISA conversion is combined to a VLES architecture that is popular in modern VLIW processors. The experimental results are presented in Section 3.4 and how others have tried to reduce code size, chip area, and power consumption of VLIW processors is discussed in Section 3.1.

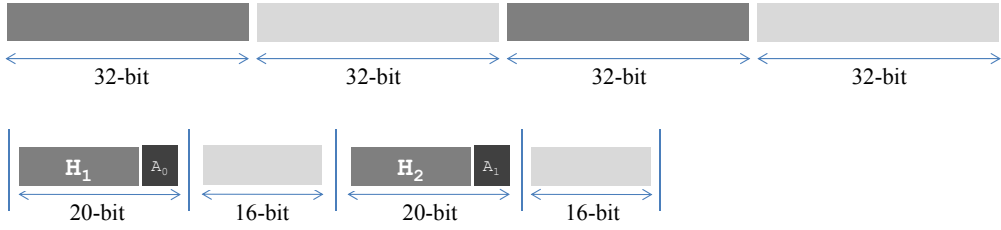


Figure 2.1 32-bit ISA vs. reduced ISA

2.1 Conceptual View

Figure 2.1 shows an example where 32-bit instructions are converted to 16-bit ones. As demonstrated in the example, some are converted nicely to *complete* 16-bit instructions. But normally, instructions as they are cannot be encoded within the 16-bit word limit, thus requiring additional bits to represent more operands. In the example, two dark colored instructions are split respectively into pairs of a 16-bit *partial* instruction and a 4-bit operand. In our approach, the register operands overflowing the word limit, which we called remote operands earlier, are stored separately into different issue slots in the VLIW code. At run time, the pair will be assembled together to form a complete instruction for execution, as illustrated in Figure 2.2.

Clearly, storing remote operands in the code may increase the overall code size, which limits the advantage of the ISA conversion. Fortunately for many cases, we find that instructions designate the same name for source and destination operands. In this work, we try to utilize this fact as much as we can to avoid creating remote operands. As in the case of `add r4, r4, r5` of Figure 2.3, we thereby were able to generate complete 16-bit instructions frequently in our experiment. But not surprisingly, most instructions still become partial instructions coupled with remote operands. To distinguish between complete and partial instructions, we set aside one bit, called *D-bit*, in the 16-bit ISA. D-bit is represented in assembly syntax as shown in Figure 2.3(c). In Section 2.2, we discuss how D-bit is decoded by hardware in the pipeline stage.

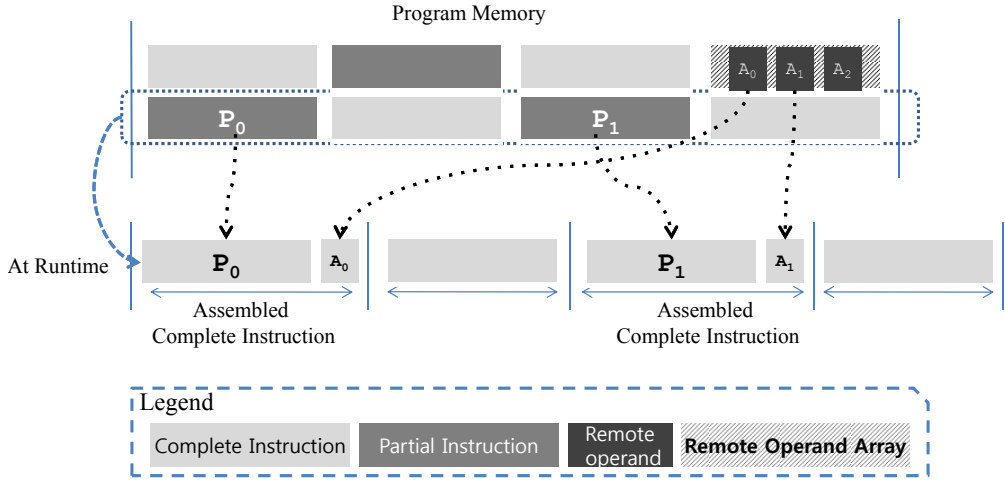
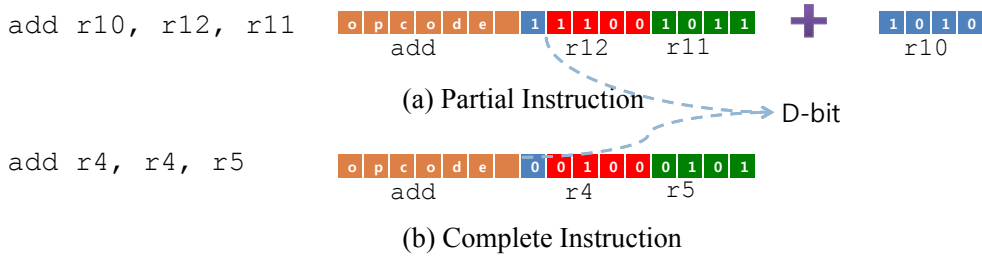


Figure 2.2 Conceptual View



	16-bit Complete Instruction	16-bit Partial Instruction
D-bit value	0	1
syntax	opcode 0, opn1, opn2	opcode 1, opn1, opn2
example	add 0,r1,r2	add 1,r1,r2
sematic	r1 = r1 + r2	remote operand = r1 + r2

(c) Assembly Syntax

Figure 2.3 Example of Partial and Complete Instructions

Once remote operands are generated, we group them to form arrays of remote operands such that each array can be mapped to an issue slot in the VLIW code. In our implementation, the slot for an array of remote operands, called a *remote operand array* (ROA) slot, reserves four bits to encode that it is a ROA slot. The remaining

twelve bits are used to store one array. Since we currently need 4 bits to encode a register operand, we can store up to three remote operands per ROA slot, as pictured in Figure 2.2. After every 32-bit instruction is converted to either a 16-bit complete or partial instruction, the compiler schedules the converted instructions into the issue slots in each VLIW packet. Unlike ordinary VLIW scheduling, our compiler needs to simultaneously schedule ROAs with other 16-bit instructions into the VLIW packets. As detailed in Section 2.3, the scheduling algorithm is designed to have ROA slots occupy the VLIW slots that otherwise would be filled with NOPs. The reason to choose NOP slots as the target place storing ROA slots is clear as mentioned in Section 1.2.1: by converting the potential NOP slots into useful ROA slots, we can avoid code size increase caused by the remote operands inside the VLIW code. Our experiment exhibits that many NOP slots commonly found in ordinary VLIW code successfully disappear in our code being replaced by ROA slots.

When we store remote operands in a ROA slot, we do so in an orderly fashion. This means that if hardware finds a partial instruction and its remote operand in a ROA slot during the decode stage, it automatically extracts the first operand in the slot. Next time the slot is retrieved for another operand, the second operand will be extracted. Therefore, the compiler should consider this hardware mechanism when it stores remote operands in ROAs. In the following sections, we will discuss in more detail how each partial instruction and its remote operand(s) are identified by hardware at run time, and how they are arranged in the code at compile time.

2.2 Architecture Design

In this section, we explain how we build the 16-bit VLIW architecture in order to execute partial instructions and remote operands as described in Section 2.1.

2.2.1 ISA Design

Our goal is not to squeeze a 32-bit instruction forcefully into a 16-bit long word, but to convert a 32-bit ISA into an equivalent 16-bit one which conserves as much information in the 32-bit ISA as possible. Our target processor [40] has to support about a hundred different operations, so in our 16-bit ISA, 8 bits are reserved for the opcode plus D-bit, leaving only 8 bits available for the operand field where we can assign register operands and an immediate operand. In this sub-section, we explain how the 32-bit ISA is mapped to a 16-bit ISA while minimizing the loss of the original 32-bit ISA semantics.

The instructions are classified into three types: R-type (register), I-type (immediate) and J-type (jump) instructions. As shown in Figure 2.4(a), an R-type instruction has the fields to specify three register operands. The 32-bit ISA has enough encoding space for all three operands. In fact, it even has large empty space, so the ISA designers often utilize the space to encode another operation; that is, they create *composite* instructions like `add-shift` which combines add and shift operations in a single instruction word. In this case, we try to capitalize on the multiple issue slots of VLIW architecture by making more than one 16-bit *basic* R-type instructions equivalent to one composite instruction. For instance, two 16-bit instructions `add` and `shift` are created and mapped to `add-shift`, and by the VLIW scheduler they will be scheduled to different issue slots according to their inter-dependences. For this, we first exclude composite instructions from the instruction set, and build a set of basic R-type instructions which constitute to be functionally equivalent to each composite instruction. Including these basic instructions, all R-type instructions are converted to either 16-bit complete instructions with two operands or partial instructions with two register operands and a 4-bit remote operand.

The I-type instruction has two register operands along with a 16-bit field to rep-

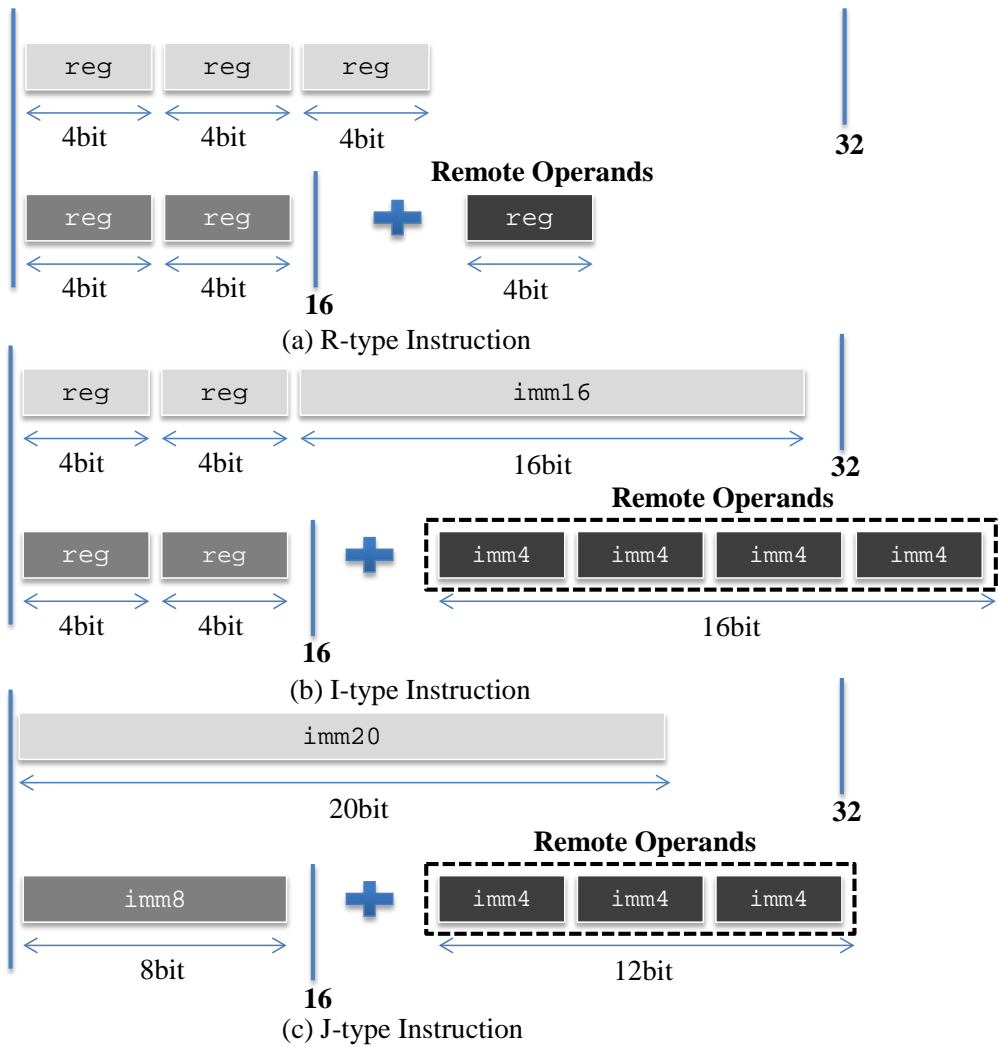


Figure 2.4 ISA Conversion

resent an immediate value. Figure 2.4(b) shows how I-type instructions are converted into the 16-bit ISA without any information loss. The 16-bit long I-type instruction is a partial instruction with two register operands, and the 16-bit immediate value becomes its remote operand that actually occupies two ROAs in our ISA design. The J-type instruction, as shown in Figure 2.4(c), requires 20 bits for its immediate address value.

We first assign an 8-bit immediate value inside the partial instruction, and render the remaining 12 bits as the remote operand, similarly to the I-type instruction. At run time, the 8-bit immediate operand and the 12-bit remote operand combine to restore the original 20-bit immediate operand.

2.2.2 Remote Operand Array Buffer

When the compiler schedules instructions and ROAs into VLIW slots, there is a strict constraint for it to obey. For more concise description, let us define P to be a partial instruction, and R to be a list of the remote operands coupled to P . Suppose that S is an issue slot that stores R . Then if the compiler schedules the ROA slot, S must be scheduled no later than P . We enforce this constraint to simplify the hardware because otherwise we must provide special storage to temporarily store and manage P until S appears. However, notice that we still need a small buffer to store remote operands appearing before their coupled partial instructions. For this we have implemented the buffer, called a *ROA buffer*. During the decode stage, a ROA slot is recognized, and all remote operands in the slot are copied to the ROA buffer. The buffer works as a rotary FIFO queue, so when new remote operands are read into the buffer, they are sequentially added at the current rear. When a partial instruction is decoded, its remote operands are removed from the front of the buffer queue. They are then attached to the instruction which then proceeds to the execution stage.

This simple rule on the ROA buffer FIFO operations works well for general cases. However, the complexity arises when multiple ROA slots are packed in the same VLIW packet, as shown in Figure 2.5(a). In this case, we need to decide which slot must be first read into the buffer. To resolve the ambiguity, we add a constraint: multiple ROA slots in a VLIW packet are processed orderly from the left to the right. Subject to this constraint, the two ROAs [3,a,b] and [c,d,8] are sequentially copied to the ROA buffer, as displayed in Figure 2.5(b) and (c). In the example, we see three

instructions with D-bit on, indicating that they are partial instructions. At cycle 0, the first partial instruction, `add r1, r2`, is decoded. As defined earlier, the first remote operand 3 is extracted from the buffer and interpreted as a register operand `r3` which is then inserted to the instruction to form a complete instruction, `add r1, r2, r3` (see Figure 2.5(d)). How a remote operand is interpreted and restored is already encoded within the opcode field, so we do not provide any special bits for this.

Figure 2.5 shows that two partial instructions appear concurrently in the same VLIW packet at cycle 1. We must be clear about which instruction should be first bound to remote operands from the ROA buffer. Similar to the case of ROA slots, multiple partial instructions are handled orderly from the left to the right. Therefore, as can be seen in Figure 2.5(e) and (f), the instruction `lw` is decoded before the instruction `mul`. Notice in Figure 2.5(e) that two pieces ('ab' and 'cd') of a remote operand are assembled together with an I-type instruction `lw`. Commonly, a partial instruction finds one remote operand in an ROA, but in principle, it may need to gather multiple pieces of its operand from consecutive arrays as in this example. The hardware automatically identifies from the opcode how remote operands must combine to be attached to the instruction. Fortunately, multiple pieces of an operand distributed over more than one ROA slots have always been concatenated in the ROA buffer before a partial instruction actually requests the operand, as exemplified in Figure 2.5. Note that, as mentioned earlier, the two pieces of a remote operand all appear before `lw` is decoded.

Once all partial instructions in the current VLIW packet are assembled with proper remote operands during the decode stage, every instruction in the packet becomes complete ready for execution. In the next subsection, we will describe the pipeline architecture for the decode stage.

2.2.3 Microarchitecture

We provide special logic for managing the ROA buffer in the pipeline architecture, as displayed in Figure 2.6.

As soon as a ROA slot is detected in the beginning of the decode stage, the remote operands in the slot are sequentially written to the buffer at the position pointed by the *write buffer pointer*. In the current implementation, the buffer pointer is incremented by word size, and one word in the ROA buffer is composed of 4 bits, as pictured in Figure 2.5. Since each slot supplies 12 bits, three is added to the address of the pointer each time a ROA slot is encountered.

When a partial instruction is decoded, its remote operands are read from the buffer. We use the *read buffer pointer* to address the first word from which the operands are fetched. After the operands are read and attached to the instruction, the number, say w , of the buffer words that have contained them is added to the buffer pointer, indicating their removal from the buffer. In our target machine, every type of partial instructions has one remote operand as classified in Figure 2.4, but in principle, they can have more than one operands. Provided that a partial instruction demands n remote operands, w is computed as follows:

$$w = \sum_{i=1}^n \lceil \frac{b_i}{4} \rceil \quad (2.1)$$

where b_i is the number of bits required to encode the i -th remote operand. For instance, w must be 1 for the R-type instruction and 4 for the I-type instruction. Computing w does not impose run time overhead as the value is statically determined for each instruction and encoded in the opcode.

Recall that if multiple ROA slots appear in a VLIW packet, they are processed serially from left to right by adjusting the write buffer pointer. As shown in Figure 2.6, to enable their operands to rush into the buffer in one cycle, we support multiple write ports respectively dedicated to each VLIW slot. Through their dedicated ports, all ROA

cycle	slot 0	slot 1	slot 2	slot 3
0	ROA 3,0xa,0xb	ROA 0xc,0xd,8	add 1,r1,r2	movi 0,r10,10
1	sub 0,r3,r4	lw 1,r5,[r10]	mul 1,r6,r7	addi 0,r10,4

buffer 

(a) an example code and an initial buffer

cycle	slot 0	slot 1	slot 2	slot 3
0	ROA 3,0xa,0xb	ROA 0xc,0xd,8	add 1,r1,r2	movi 0,r10,10
1	sub 0,r3,r4	lw 1,r5,[r10]	mul 1,r6,r7	addi 0,r10,4

buffer 

(b) ROA 3,0xa,0xb writes remote operands into the buffer

cycle	slot 0	slot 1	slot 2	slot 3
0	ROA 3,0xa,0xb	ROA 0xc,0xd,8	add 1,r1,r2	movi 0,r10,10
1	sub 0,r3,r4	lw 1,r5,[r10]	mul 1,r6,r7	addi 0,r10,4

buffer 

(c) ROA 0xc,0xd,8 writes remote operands into the buffer

cycle	slot 0	slot 1	slot 2	slot 3
0	ROA 3,0xa,0xb	ROA 0xc,0xd,8	add 1,r1,r2	movi 0,r10,10
1	sub 0,r3,r4	lw 1,r5,[r10]	mul 1,r6,r7	addi 0,r10,4

buffer 

(d) add 1,r1,r2 reads one remote operand from the buffer

cycle	slot 0	slot 1	slot 2	slot 3
0	ROA 3,0xa,0xb	ROA 0xc,0xd,8	add 1,r1,r2	movi 0,r10,10
1	sub 0,r3,r4	lw 1,r5,[r10]	mul 1,r6,r7	addi 0,r10,4

buffer 

(e) lw 1,r5,[r10] reads four remote operands from the buffer

cycle	slot 0	slot 1	slot 2	slot 3
0	ROA 3,0xa,0xb	ROA 0xc,0xd,8	add 1,r1,r2	movi 0,r10,10
1	sub 0,r3,r4	lw 1,r5,[r10]	mul 1,r6,r7	addi 0,r10,4

buffer 

(f) mul 1,r6,r7 reads one remote operand from the buffer

Figure 2.5 ROA buffer operations

slots can be written to the buffer concurrently. Likewise, multiple partial instructions can access their operands within a cycle through their dedicated read ports. The design issue here is how to manage the buffer pointers in the presence of multiple ROA slots or partial instructions such that the corresponding remote operands are written or read concurrently and yet orderly from left to right. Figure 2.7 presents the combinational logic that manages the pointers. The leftmost slot in our architecture is slot 0. So the ROA slot (or partial instruction) in slot 0 has the highest priority of accessing the first word in the ROA buffer. Assume that both slot 0 and slot 1 are ROA slots, and the write buffer pointer has a value p_0 . Then, the first slot accesses the word addressed at p_0 in the buffer, and the second does the word addressed at $p_0 + x$ where x is three in our architecture as explained before. If slot 0 is not a ROA slot in the example, x becomes zero. The write buffer pointer will be updated at the end of the cycle, and the value depends on the number of ROA slots encountered during this cycle, as can be seen in Figure 2.7.

In our design, we also allow multiple partial instructions and ROA slots to appear in the same packet. If they both do not have any inter-dependence (i.e., any of the instructions demands no remote operands from the ROA slots), they can be processed independently following the procedure listed just above. But sometimes, we need to handle the case where a partial instruction seeks its remote operands from the ROA slot(s) in the same packet. To tackle this case, we enforce the restriction that the write to the ROA buffer should occur in the first half of the clock cycle and the read from the buffer occur in the second half. In accordance with this hardware restriction, the compiler schedules a ROA slot to the left of a partial instruction dependent on the slot if they are to be placed in the same VLIW packet.

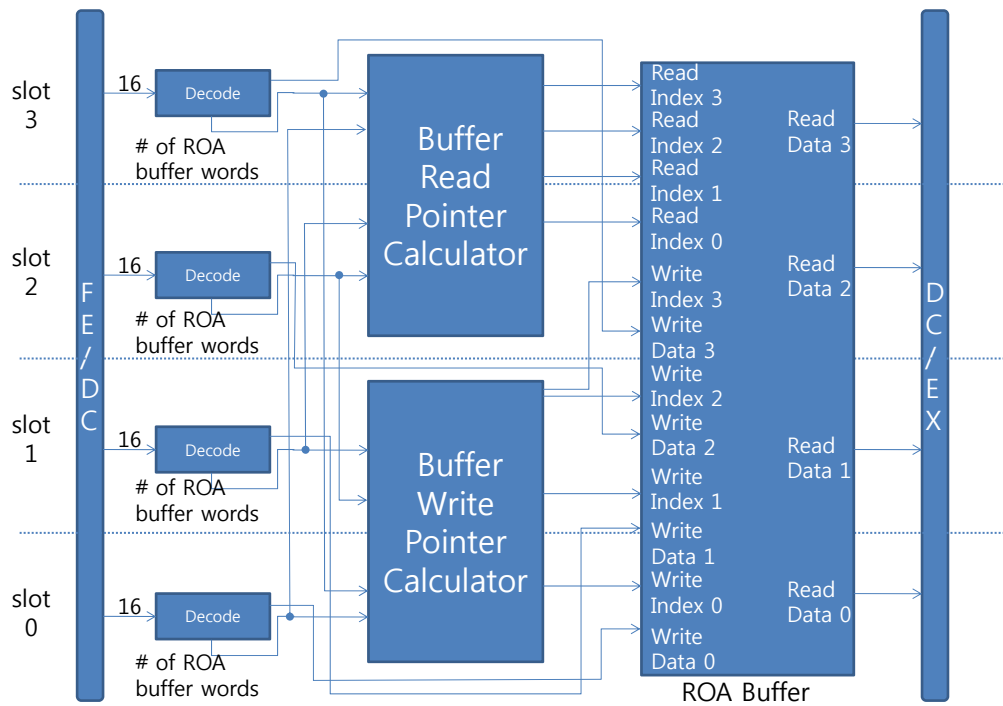


Figure 2.6 Decode Stage with ROA buffer

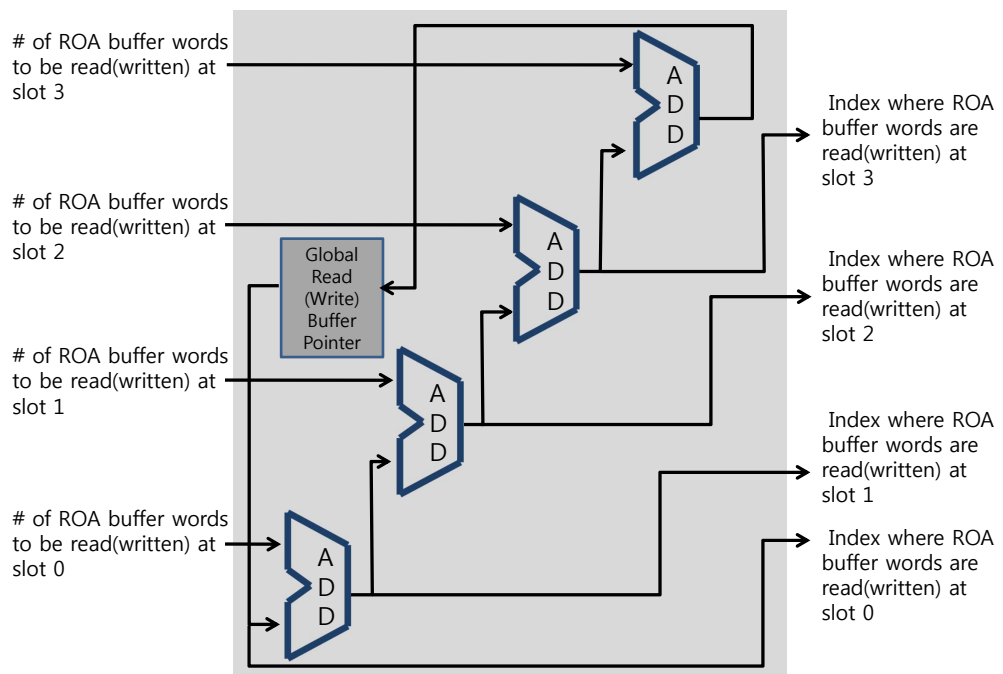


Figure 2.7 Read/Write buffer pointer management

2.3 Compiler Support

In order to build instructions using the ROA slot, we have implemented a code generation framework where a sequence of 4-way 16-bit VLIW packets including partial instructions and ROA slots is generated through three phases. In the first phase, the source code is compiled to the code in a low-level intermediate representation. As exemplified in Figure 2.8(a), the statement $x = y + z$ in the source code is translated to an assembly-like instruction `add r5, r1, r2`. All the resulting instructions are complete in that their operands are all explicitly represented. In the subsequent phases, VLIW packets are generated, and partial instructions as well as ROA slots are also generated at some point to be scheduled somewhere within the packets.

The decision we should make here is which must be generated first between VLIW packets and ROA slots. One strategy would be to generate VLIW packets first, and to schedule ROA slots later, as demonstrated in Figure 2.8. The VLIW scheduler constructs a sequence of VLIW packets each of which has four issue slots, as shown in Figure 2.8(b). At this point, the instructions are assumed to be in a 32-bit word format. Therefore, they are not yet executable on our target processor that provides the 16-bit hardware word limit. In the following phase, we analyze the operand field of each instruction in the VLIW code in order to select candidates for partial instructions. Then, we create ROAs for those selected ones and build ROA slots which are in turn placed at appropriate positions (preferably the slots filled with NOPs) in the code. A major downside of this strategy is clearly visible from the output code in Figure 2.8(c), where an extra packet is added solely to accommodate the newly created ROA slots. Even if the code in Figure 2.8(b) originally has three NOP slots, the final code fails to replace any of them by the ROA slots. This is due to the constraints (stated in Section 2.2) on the placement of an ROA slot relative to the partial instruction whose remote operand is present in the slot. Subject to one of the constraints, for example

```

add   r5,r1,r2
mov   r12,r13
addi  r14,r13,0xabcd
addi  r10,r10,1
mul   r5,r5,r6
add   r7,r12,r3
mul   r9,r6,r8
add   r1,r10,r13
sub   r11,r14,r12

```

(a) an example of a single-issue code

cycle	Slot 0	Slot 1	Slot 2	Slot 3
0	add r5,r1,r2	mov r12,r13	addi r14,r13,0xabcd	addi r10,r10,1
1	mul r5,r5,r6	add r7,r12,r3	mul r9,r6,r8	add r1,r10,r13
2	sub r11,r14,r12			

(b) a code after VLIW scheduling

cycle	Slot 0	Slot 1	Slot 2	Slot 3
0	ROA 5,0xa,0xb	ROA 0xc,0xd,7	ROA 9,1,11	
1	add 1,r1,r2	mov 0,12,r13	addi 1,r14,r13	addi 0,r10,1
2	mul 0,r5,r6	add 1,r12,r3	mul 1,r6,r8	add 1,r10,r13
3	sub 1,r14,r12			

(c) a code after converting 32-bit ISA into 16-bit ISA

Figure 2.8 An example of code generation

in Figure 2.8(b), the remote operand `r5` of an instruction `add r5, r1, r2` cannot be placed after cycle 0. Unfortunately, there is no NOP slot at cycle 0 that can be replaced by the ROA slot with `r5`. Thus, as shown in Figure 2.8(c), the ROA slot is placed into a VLIW packet that is newly created and inserted before the instruction. The addition of a new packet results in run time increase by 25%. Of course, we can still achieve 50% reduction in code size since the instruction word length is halved from 32 bits to 16 bits. However, we could have achieved further reduction if we better utilized NOPs at cycle 2 when ROA slots were placed inside the code.

To mend this problem, in our implementation, which is presented in Figure 2.9, we choose the other strategy where 16-bit partial instructions and ROA slots are generated before VLIW packets are constructed. In fact, Figure 2.13 presents the actual output code from our compiler. We can see that NOPs all disappear after being replaced by ROAs. The code size is now exactly halved, which is the ideal case we could achieve

thru the ISA conversion. A main benefit of this strategy is that when we schedule instructions to construct VLIW packets, we can simultaneously consider the above-mentioned constraint. In this section, we will explain our code generation algorithm in Figure 2.9.

2.3.1 16-bit Instruction Generation

Figure 2.10 summarizes the overall flow of our compilation framework. This subsection corresponds to the second phase of our framework. In this phase, we generate 16-bit executable instructions from the 32-bit pseudo instructions. For this, we check the operands of each instruction in order to decide whether or not they should become remote operands. If an instruction is found to require remote operands, we divide it into a partial instruction and remote operands, subsequently assigning an ROA for the remote operands. Figure 2.11 presents an example of the instruction generation process.

In Figure 2.11(a), we see the singles-issue 32-bit code whose instructions include all their operands. This code has been produced in the earlier phase and given as input to this phase. As depicted in Figure 2.11(b), remote operands of each instruction from the input code are identified. For instance, `addi r14, r13, 0xabcd` is an I-type instruction which includes a 16-bit immediate operand with two register operands. So, the immediate operand will be split from the instruction and stored as a remote operand. After all remote operands of each instruction are identified, they are clustered into arrays of 12 bits length. Since each ROA buffer word is 4 bits long, a single array of remote operands consists of three words, as already discussed earlier. Figure 2.11(c) shows how we cluster remote operands in the example code. As the last step, a ROA slot is created for each array in the code (see Figure 2.11(d) for example), and inserted into the code. The resulting code will be given as input to our VLIW scheduler in the next phase discussed below. There is basically no restriction on the place where an

Input : A single-issue 32-bit instruction code C
Output : A multiple-issue code with ROA slots

Build a control flow graph G from C
Let $G=(N,E)$ where $N = \{n_i \mid n_i \text{ is a basic block}\}$,
 $E = \{e_{ij} \mid \exists e_{ij} \text{ if there is a control flow } n_i \rightarrow n_j\}$

```

01: for each  $n_i \in G$ 
02:    $current\_inst = NULL$ ;
03:    $unresolved\_inst = NULL$ ;
04:    $input\_code\_seq = n_i.get\_code\_seg()$ ;
05:    $output\_code\_seq = NULL$ ;
06:    $resolved\_inst\_seq = NULL$ ;
07:    $ROA\_buffer = NULL$ ;
08:    $temp\_ROA\_buffer = NULL$ ;
09:    $ROA\_slot = NULL$ ;
10:    $ROA\_slot\_generated = false$ ;
11:   while  $!input\_code\_seq.empty() \parallel unresolved\_inst \neq NULL$  do
12:     if  $unresolved\_inst \neq NULL$  then
13:        $current\_inst = unresolved\_inst$ ;
14:        $unresolved\_inst = NULL$ ;
15:     else
16:        $current\_inst = input\_code\_seq.get\_front()$ ;
17:        $buffer\_words = convert\_32bit\_to\_16bit\_and\_get\_ROA\_buffer\_words(current\_inst)$ ;
18:        $temp\_ROA\_buffer.insert(buffer\_words)$ ;
19:     fi
20:     while  $!temp\_ROA\_buffer.empty()$  do
21:        $ROA\_buffer.insert(temp\_ROA\_buffer.get\_front())$ ;
22:       if  $ROA\_buffer.size() \% 3 = 0$  then
23:          $ROA\_slot = generate\_ROA\_slot(ROA\_buffer)$ ;
24:          $ROA\_slot\_generated = true$ ;
25:         if  $ROA\_buffer.full()$  then
26:            $ROA\_buffer.clear()$ ;
27:         fi
28:         break
29:       fi
30:     od
31:     if  $temp\_ROA\_buffer.empty()$  then
32:        $resolved\_inst\_seq.insert(current\_inst)$ ;
33:     else
34:        $unresolved\_inst = current\_inst$ ;
35:     fi
36:     if  $ROA\_slot\_generated == true$  then
37:        $output\_code\_seq.insert(ROA\_slot)$ ;
38:        $output\_code\_seq.insert(resolved\_inst\_seq)$ ;
39:        $ROA\_slot\_generated = false$ ;
40:     fi
41:   od
42: end
43: for each  $n_i \in G$ 
44:    $build\_data\_dependence\_graph(n_i)$ ;
45:    $apply\_list\_scheduling(n_i)$ ;
46: end

```

Figure 2.9 Algorithm for a multi-issue code generation

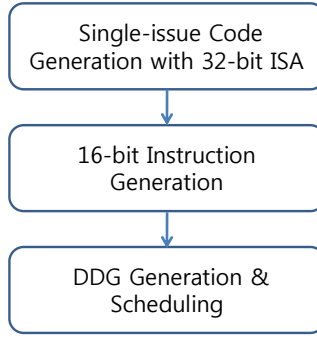


Figure 2.10 Compilation Framework

ROA slot is inserted inside this code, except, as illustrated in Figure 2.11(d), that each slot should be inserted before the partial instructions whose remote operands exist in the slot. This restriction facilitates the construction of data dependence graphs (DDGs) where dependence edges flow from remote operands to their coupled instructions. In the following subsection, we will discuss how DDGs are constructed and used by the VLIW scheduler to generate multi-issue code with ROA slots from the single issue code produced in this phase.

2.3.2 DDG Construction & Scheduling

In order to determine whether rearranging instructions in a certain way preserves their own behavior, we need a DDG, which contains nodes that represent instructions and edges that represent data dependencies between instructions. Each edge is labeled with the latency of dependence which is the number of clock cycles that needs to elapse before the pipeline may proceed with the target instruction without stalling. The first task in this phase is to construct a DDG for each basic block of code like that in Figure 2.11(d). Generally, constructing a DDG is trivial, but in our processor ROA slots need special treatment. So we first introduce how ROA slots are added into DDG. In Figure 2.12, the DDG is drawn for the example code in Figure 2.11(d). In the graph, we represent the dependencies among ROA slots and partial instructions in the solid

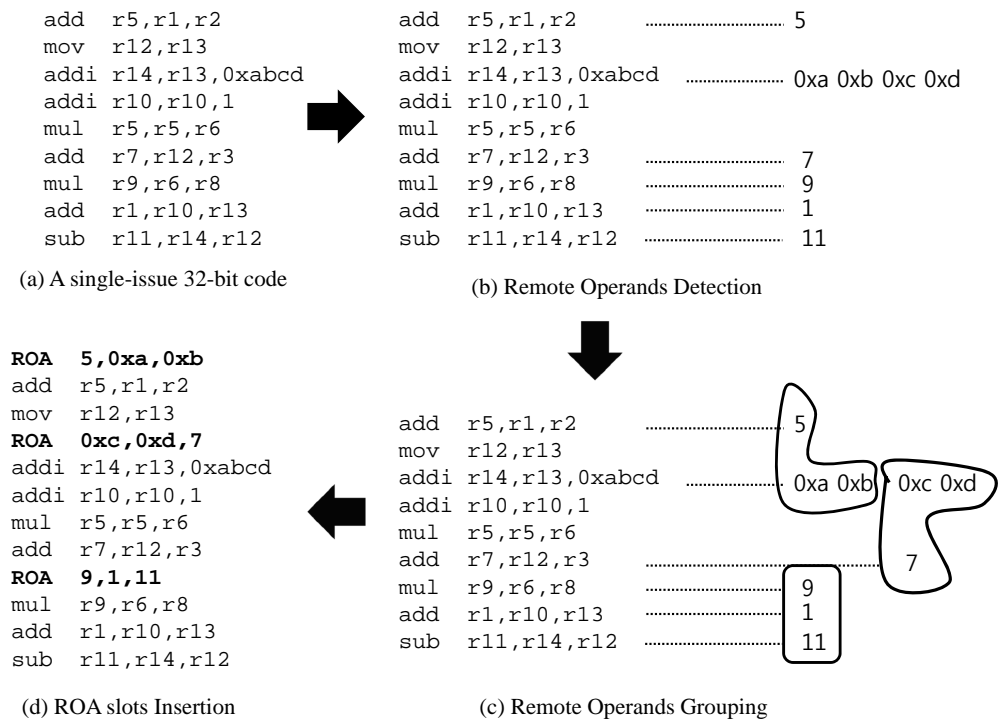


Figure 2.11 An example of 16-bit Instruction generation

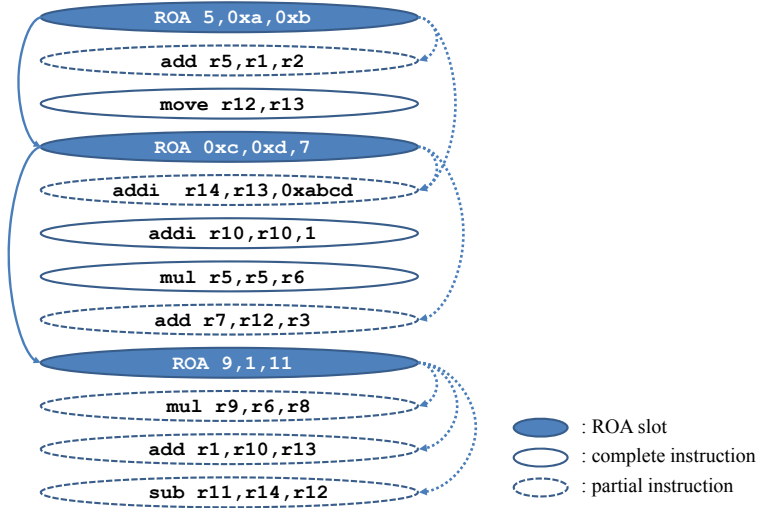


Figure 2.12 Data Dependence Graph

or dotted lines. For convenience's sake, we only represent dependencies related with ROA slots.

A solid line here denotes the dependency between two ROA slots that is used to sequentialize the order in which the slots are read into the ROA buffer. A dotted line denotes the dependency from a ROA slot to a partial instruction. The latency of these two types of dependencies is zero while the latency of dependencies between ordinary instructions is one or higher. This is because two ROA slots can be scheduled in the same VLIW packet, and also because a remote operand in the ROA slot and its coupled partial instruction are combined and executed at the same cycle. Let us remind that each issue slot has the different buffer access priority as depicted in Figure 2.7. Therefore, when a ROA slot and its dependent partial instructions are assigned at the same cycle, we have to make sure that the ROA slot is assigned at the higher priority issue slot (i.e., the leftmost possible one) in the packet than any of the instructions.

The final multiple-issue code including ROA slots is shown in Figure 2.13, where the schedule order in the code is determined fully reflecting not only the dependen-

cycle	slot 0	slot 1	slot 2	slot 3
0	ROA 5,0xa,0xb	add 1,r1,r2	mov 0,r12,r13	ROA 0xc,0xd,7
1	addi 1,r14,r13	add 0,r10,1	mul 0,r5,r6	add 1,r12,r3
2	ROA 9,1,11	mul 1,r6,r8	add 1,r10,r13	sub 1,r14,r12

Figure 2.13 Generated Multiple-issue Code with ROA slot

cies involving ROA slots in Figure 2.12 but also those among ordinary instructions. A sequence of VLIW packets is fabricated from each fragment of the code like this example which corresponds to a single basic block represented in DDGs. This implies that during execution, ROA slots become aligned in the ROA buffer at every entry of the basic blocks, which enables the hardware to automatically increment the write buffer pointer by three as described in Section 2.2.

2.4 VLES(Variable Length Execution Set) Architecture with a Reduced Bit-width Instruction Set

Recent VLIW processors adopt a VLES (variable length execution set) architecture to provide compact code density [3] [39]. As stated in Section 1.2.1, a fair number of NOPs exist in VLIW code due to the lack of ILP (instruction level parallelism), which incurs loose code density. In the VLES architecture, these NOPs are not explicitly present in the VLIW code and thus the code density can be substantially improved. However, existing VLES architectures adopt only a 32-bit ISA or partially support a 16-bit ISA under the 32-bit ISA like normal VLIW architectures. Because of this 32-bit ISA, the VLES architecture also suffers from the wide instruction word length. In common with the case of normal VLIW architectures, if the 32-bit ISA is fully converted to the equivalent 16-bit one in VLES architectures, problems due to the long instruction word could be resolved. Therefore, we need to know how the features of VLES architectures could affect our proposed ISA conversion.

In VLIW architectures, remote operands could be possibly allocated to NOPs if

there exist. Otherwise, the remaining remote operands should be inserted to newly generated VLIW packets, which incur the increase of code size. In contrast, there is no NOP in VLES architectures. Thus, all remote operands become overheads in terms of the code size in VLES architectures. However, since the amount of code size reduction due to the instruction bit-width reduction is bigger than that of code size increase due to the remote operands, we could eventually achieve the code size reduction in VLES architectures with a reduced bit-width instruction set even if the rate of final code size reduction is smaller than in VLIW architectures. In our experiment, we will show how the ISA conversion affects the code size in VLES architectures.

Note that the removal of NOPs in VLES architectures cannot prevent functional units from being idle at run time. This is because VLES architectures cannot overcome the fundamental lack of ILP in a program. In normal VLIW architectures that do not adopt a VLES architecture, the lack of ILP is represented as NOPs in the code at compile time, and then these NOPs incur idle slots at run time. In the VLES architectures, although there does not exist NOPs in the code, the lack of ILP is also reflected as the idle slots at run time since the lack of ILP is encoded in a different manner as will be explained in the following subsection. Therefore, the concept of utilizing the idle slots for remote operands is identically applied to both VLIW and VLES architectures, implying that the VLES architectures do not affect our proposed ISA conversion with the perspective of runtime behavior. The experimental result in terms of runtime will also be presented in Section 6.

2.4.1 Architecture Design

In VLES architectures, instructions are read from program memory every cycle in bundles of a fixed size called a *fetch packet*. Then, VLES architectures extract instructions executable in parallel, called an *execute packet*, from the fetch packet and dispatch them to functional units. Since the number of instructions in an execute packet is not

15	9	8	6	5	4	3	0
prefix				don't care		# of parallel instructions	issue slot assignment

(a) encoding of a prefix instruction

syntax	# of parallel instructions	issue slot assignment
prefix #1, 0	1	slot 0
prefix #1, 1	1	slot 1
prefix #1, 2	1	slot 2
prefix #1, 3	1	slot 3
prefix #2, 0	2	slot 0, 1
prefix #2, 1	2	slot 0, 2
prefix #2, 2	2	slot 0, 3
prefix #2, 3	2	slot 1, 2
prefix #2, 4	2	slot 1, 3
prefix #2, 5	2	slot 2, 3
prefix #3, 0	3	slot 0, 1, 2
prefix #3, 1	3	slot 0, 1, 3
prefix #3, 2	3	slot 0, 2, 3
prefix #3, 3	3	slot 1, 2, 3

(b) meaning of each prefix instruction according to its corresponding encoding

Figure 2.14 A prefix instruction for our VLES architecture

consistent for each cycle, one instruction of a fetch packet is designated to store configuration information for an execute packet in the existing VLES architectures [3] [39]. In this dissertation, this kind of an instruction is called a *prefix instruction*. In the experiment, it is given a 4-way VLES processor with a 32-bit ISA that has also a 32-bit prefix instruction. To convert a 32-bit VLES architecture to an equivalent 16-bit one, a 32-bit prefix instruction should be able to be reduced to a 16-bit one. Figure 2.14 represents an encoding format and semantics of our 16-bit prefix instruction. In our implementation, the maximum number of instructions executed in parallel is four (4-way VLES architecture). The prefix instruction is assigned at the first slot in a fetch packet when the number of instructions executable in parallel is less than four in our VLES architecture. On the other hand, the prefix instruction is not encoded in a fetch packet if four instructions are able to be executed in parallel. Figure 2.14(b) represents

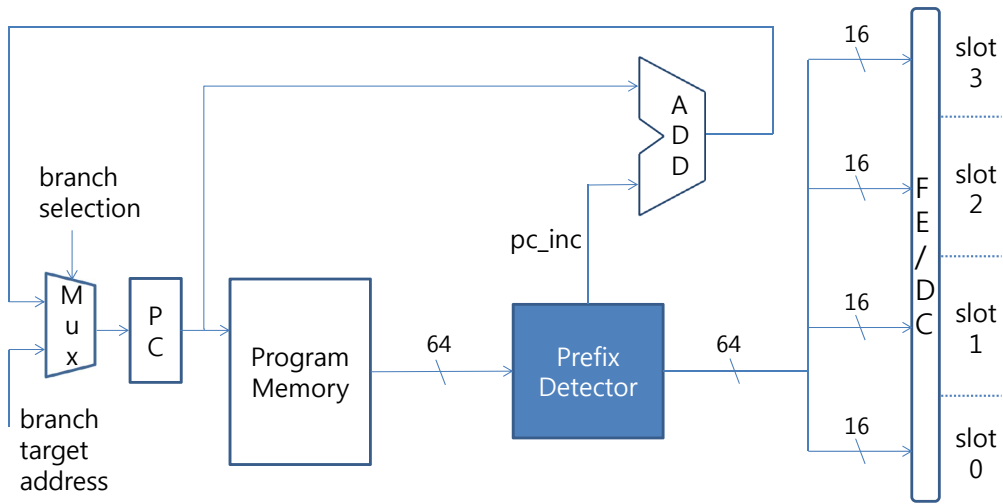
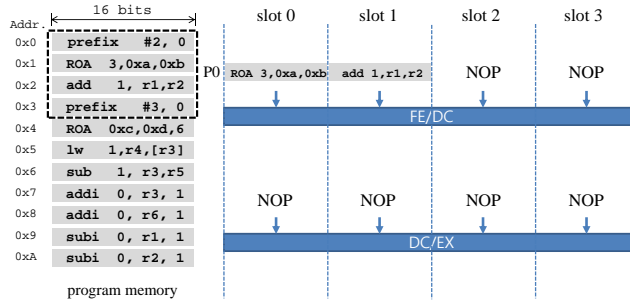


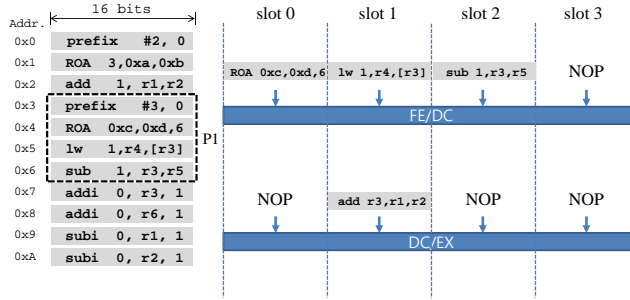
Figure 2.15 A fetch logic for our VLES architecture

the number of instructions executable in parallel and the issue slot assignment for each instruction of them according to encoding of the prefix instruction. Figure 2.15 shows a fetch logic for our VLES architecture. For each fetch packet, a *prefix detector* checks whether or not the fetch packet has a prefix instruction, and then it determines the value of *pc_inc* and an execute packet for being delivered to the decode stage where partial instructions and their corresponding remote operands are combined to reconstruct the original 32-bit instructions in the same way as explained in Section 2.2. *pc_inc* is used to compute *next PC* (program counter). For example, if an execute packet consists of two parallel instructions, *pc_inc* becomes two and *next PC* is calculated to be *current PC* + 2.

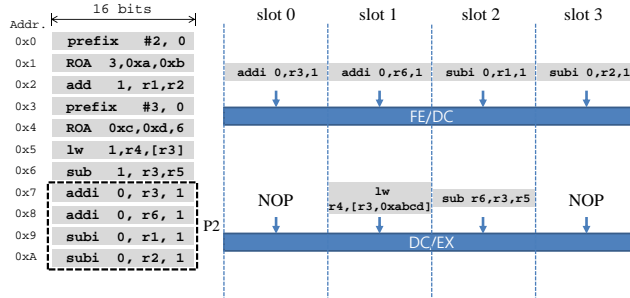
To help understand behavior of the VLES architecture, Figure 2.16 presents an example how our VLES architecture works with a prefix instruction. A fetch packet consisting of four consecutive instructions is fetched every cycle from program memory in our VLES architecture. At cycle 0, the first instruction in *P0* is a prefix instruction as shown in Figure 2.16(a). It indicates that next two instructions are executable in par-



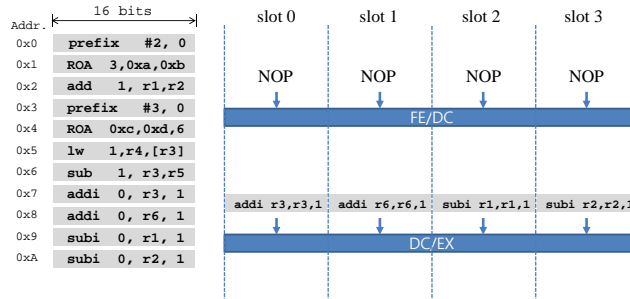
(a) The state of the pipeline architecture at the end of cycle 0



(b) The state of the pipeline architecture at the end of cycle 1



(c) The state of the pipeline architecture at the end of cycle 2



(d) The state of the pipeline architecture at the end of cycle 3

Figure 2.16 An example for behavior of our VLES architecture (P_t represents a fetch packet at cycle t ($t=0,1,2,\dots$), FE/DC denotes the pipeline register between fetch(FE) and decode(DC) stages, DC/EX denotes the pipeline register between decode(DC) and execution(EX) stages.)

allel and each of them should be sent to slot 0 and slot 1, respectively. The instruction of slot 0 is a ROA and that of slot 1 is a partial instruction coupled to the ROA, so the partial instruction, `add r1, r2`, becomes complete with a remote operand, `r3`, at the decode stage and then it is dispatched to the execution stage (see Figure 2.16(b)). Note that the fourth instruction in *P0* is also a prefix instruction, but it is ignored at cycle 0 since it contains information for an execute packet of the next cycle. Thus, the value of *pc_inc* becomes three and since *current PC* is `0x0` at cycle 0, then *next PC* is computed to be `0x3`. Consequently, *P1* is read from the address, `0x3`, at cycle 1 as shown in Figure 2.16(b). Similar to the case of *P0*, *P1* also has a prefix instruction which has configuration information for the execute packet at cycle 1. In the case of *P1*, two partial instructions are dispatched to the execution stage after being complete with their corresponding remote operands at the decode stage (see Figure 2.16(b) and (c)). In Figure 2.16(c), we can see that there is no prefix instruction in *P2*, indicating that all four instructions in *P2* can be executed in parallel. Since they are all complete instructions, they are intactly dispatched to the execution stage without combining remote operands as shown in Figure 2.16(d).

2.4.2 Compiler Support

Figure 2.17 describes a code transformation algorithm for our 16-bit VLES architecture. It takes as input a 16-bit VLIW code generated by the algorithm in Figure 2.9. Obviously, an input code has NOPs as well as ROAs. By the algorithm in Figure 2.17, an output code is generated with prefix instructions and without NOPs. Figure 2.18 shows an example for transformation from VLIW code to VLES code. The first four consecutive instructions (address `0x0-0x3`) are changed to one prefix instruction and two valid instructions. The second VLIW packet (address `0x4-0x7`) is transformed to one prefix instruction and three valid instructions. Two NOPs and one NOP are removed from each packet, respectively. However, since the third four consecu-

```

Input : A multiple-issue 16-bit instruction code  $C$  where there exist ROAs and NOPs
Output : A multiple-issue 16-bit instruction code with ROAs and prefix instructions and without NOPs
An input code  $C$  has a control flow graph  $G$ 
Let  $G=(N,E)$  where  $N = \{n_i \mid n_i \text{ is a basic block}\}$ ,
 $E = \{e_{ij} \mid \exists e_{ij} \text{ if there is a control flow } n_i \rightarrow n_j\}$ 
01: for each  $n_i \in G$ 
02:    $current\_VLIW\_packet = \text{NULL};$ 
03:    $new\_VLIW\_packet = \text{NULL};$ 
04:    $input\_VLIW\_packet\_seq = n_i.get\_VLIW\_packet\_seq();$ 
05:    $output\_VLIW\_packet\_seq = \text{NULL};$ 
06:   while  $!input\_VLIW\_packet\_seq.empty()$  do
07:      $current\_VLIW\_packet = input\_VLIW\_packet\_seq.get\_front();$ 
08:      $valid\_inst\_seq = \text{NULL};$ 
09:      $valid\_inst\_slot\_seq = \text{NULL};$ 
10:      $current\_slot\_number = 0;$ 
11:     while  $!current\_VLIW\_packet.empty()$  do
12:        $current\_inst = current\_VLIW\_packet.get\_front();$ 
13:       if  $current\_inst \neq \text{NOP}$  then
14:          $valid\_inst\_seq.insert(current\_inst);$ 
15:          $valid\_inst\_slot\_seq.insert(current\_slot\_number);$ 
16:       fi
17:        $current\_slot\_number++;$ 
18:     od
19:     if  $valid\_inst\_seq.size() \neq total\_issue\_slot\_count\_of\_VLIW\_architecture$  then
20:        $prefix\_inst = generate\_prefix\_inst(valid\_inst\_seq.size(), valid\_inst\_slot\_seq);$ 
21:        $new\_VLIW\_packet.insert(prefix\_inst);$ 
22:        $new\_VLIW\_packet.insert(valid\_inst\_seq);$ 
23:        $output\_VLIW\_packet\_seq.insert(new\_VLIW\_packet);$ 
24:     else
25:        $output\_VLIW\_packet\_seq.insert(current\_VLIW\_packet);$ 
26:     fi
27:   do
28: end

```

Figure 2.17 Code transformation algorithm for our VLES architecture

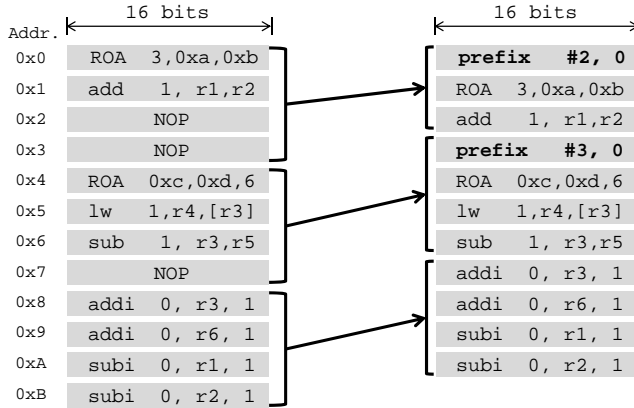


Figure 2.18 An example of code transformation for our VLES architecture

tive instructions (address 0x8–0xB) are all valid instructions, there is nothing to be transformed. Note that two bytes code size reduction is achieved in this example, which makes it possible to provide compact code density.

2.5 Experiments

2.5.1 Setup

The goal of our experiments is to estimate and validate the effectiveness of our proposed ISA conversion by comparing the original 32-bit ISA VLIW processor to the converted 16-bit one. The 32-bit ISA VLIW processor is adopted from one of the diverse processor models offered by Processor Designer from Synopsys [40]. Based on architecture description language LISA [40], each processor can be modeled to generate software tools such as assembler, linker and simulator in Processor Designer. With more detailed description, a hardware description language (HDL) code is also generated for the corresponding architecture, which is synthesizable at the gate level by using Design Compiler, a commercial synthesis tool from Synopsys. Using Processor Designer, we have implemented the converted 16-bit ISA VLIW processor from the selected 32-bit one. Note that reducing bit-width is only applied to the instruction bits

not the data bits. Thus, after the conversion, the data word length remains the same 32 bits as the original architecture, from which the behavior of a partial instruction with remote operands could be the same as that of an original 32-bit instruction at the execution stage and the next pipeline stages. Another important issue in the design of the converted processor is to determine the size of a ROA buffer. It depends on the maximum size of remote operands for a partial instruction. In the selected 32-bit ISA VLIW processor, a 16-bit immediate field is a maximal size of remote operand, which requires four ROA buffer words as explained in Section 2.2. Since a ROA slot writes three words into the buffer, the ROA buffer should store the contents of two ROA slots to make four buffer words read by the partial instruction at a time. Consequently, the size of the ROA buffer could be set to 6. Note that our decision about the ROA buffer size is made without loss of generality, even the size is quite small. In a 32-bit instruction bit-width, a remote operand cannot be composed of more than eight ROA buffer words, which can be covered by setting the ROA buffer size to 9. In other words, the upper bound for the size of the ROA buffer is 9. Of course, we can design the ROA buffer without limitations for the size. But, in case of the ROA buffer with a larger size, it will cause a much longer time to access the ROA buffer at the decode stage. Therefore, we need to limit the ROA buffer size as small as possible under the upper bound to avoid undesirable delay due to the ROA buffer access. Further, we also compare an existing 32-bit ISA VLES processor and the corresponding 16-bit one after the ISA conversion. The 32-bit ISA VLES processor is also given by Synopsys Processor Designer. It has the same ISA with the above 32-bit VLIW processor except that it should additionally support a 32-bit prefix instruction for a VLES architecture. Similarly, the converted 16-bit VLES processor is also identical to the above converted 16-bit VLIW processor except for its 16-bit prefix instruction. As a result, total four processors are evaluated in our experiments. The information about four VLIW processors is listed in Table 2.1. The compiler for each processor is generated based on a retargetable com-

Table 2.1 Four processors used in the experiments

32-bit ISA VLIW proces- sor (32VLIWP)	16-bit ISA VLIW proces- sor (16VLIWP)	32-bit ISA VLES proces- sor (32VLESP)	16-bit ISA VLES proces- sor (16VLESP)
32-bit instruction set, 32-bit data word length, five stages of pipeline (IF,DC,EX,MEM,WB)	16-bit instruction set, 32-bit data word length, five pipeline stages with a modified DC stage, a ROA buffer	32-bit instruction set, 32-bit data word length, five stages of pipeline (IF,DC,EX,MEM,WB)	16-bit instruction set, 32-bit data word length, five pipeline stages with modified FE and DC stages, a ROA buffer

piler platform, SoarGen [41]. Given the ISA of a target processor architecture written in architecture description language, SoarDL [41], SoarGen can generate the compiler for the target processor. In order to generate the code that could exploit our proposed architecture, the algorithm explained in Section 2.3 is implemented in the compiler for the converted 16-bit ISA VLIW processor. Further, the algorithm described in Section 2.4 is also implemented for the converted 16-bit ISA VLES processor. we have tested a set of benchmarks, Livermore Loops [42] and DSPstone [43], to evaluate the effectiveness of our approach. Livermore Loops is a benchmark suite for parallel architectures and consists of a set of loop kernels in numerically intensive applications. DSPstone is kernel benchmarks consisting of code fragments or functions which are commonly used in DSP algorithms. To estimate the overhead of our proposed architecture, the cell area and clock cycle time are measured. To show the efficiency of our proposed architecture, the analysis results of the code size and energy consumption are also represented in the next subsection.

2.5.2 Results

Hardware cost

The synthesizable HDL codes of four processors are generated by HDL generator in Synopsys Processor Designer [40]. Using the HDL code as input, Synopsys Design Compiler gives the logic synthesis result as shown in Table 2.2. The total cell area is the sum of combinational area and non-combinational area. Compared to that of 32VLIWP, the total cell area of 16VLIWP is decreased by about 10%. First, the combinational area is decreased since the decoding logic in 16VLIWP is simplified due to the reduced instruction bit-width, even though the pointer calculation logic to read and write the ROA buffer incurs overhead. Second, since the pipeline register in 16VLIWP stores reduced instruction bits, the required size for the pipeline register in 16VLIWP can be smaller than 32VLIWP. Thus, the non-combinational area is also decreased in 16VLIWP. In the case of VLES processors, the cell area is also reduced by the same token. The amount of the reduction of the cell area is about 11%. Note that each VLES processor has larger cell area than the same instruction bit-width VLIW processor due to the logic related to a prefix instruction.

Table 2.3 shows the clock cycle time of each processor. This result is also gained from Synopsys Design Compiler. The clock cycle time of 16VLIWP is about 0.8% longer than 32VLIWP. Similarly, in the case of VLES processors, the clock cycle time of 16VLESP is about 1.8% longer than 32VLESP. These clock cycle time increases are caused by computing the pointer values for reading and writing from/to the ROA buffer through the pointer calculation logic and by accessing the ROA buffer, which makes the decode stage become a critical path for determining the clock cycle time.

Code size

The code size for each processor is the size of program memory used to store the binary code. Figure 2.19 shows the code size of each processor normalized to that of

Table 2.2 Cell area of each processor (unit : μm^2)

	32VLIWP	16VLIWP	32VLESP	16VLESP
Combinational area	550671.26778	497965.86579	572548.8681	512816.98388
Non-combinational area	79740.07998	67408.65743	80005.62887	67758.69914
Total cell area	630411.34777	565374.52322	652554.4969	580575.68302

Table 2.3 Clock cycle time of each processor (unit : ns)

	32VLIWP		16VLIWP		32VLESP		16VLESP	
Point	Incr.	Path	Incr.	Path	Incr.	Path	Incr.	Path
Clock clk_main(rise edge)	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
Library setup time	-0.05	1.95	-0.06	1.94	-0.05	1.95	-0.07	1.93
Data required time		1.95		1.94		1.95		1.93
Data arrival time		-3.71		-3.74		-3.79		-3.86
Slack		-1.76		-1.80		-1.84		-1.93

32VLIWP. The average rate of the code size reduction from 32VLIWP to 16VLIWP is about 41%. In the ideal case where every ROA slot in the code successfully substitutes for NOPs (see Figure 2.13), the code size could be reduced ideally by a half. However, for many cases, ROA slots require extra VLIW packets. For example, when a loop kernel has a high degree of parallelism, the resulting VLIW packets do not contain an enough number of NOP slots that can be replaced by ROA slots. Then, the ROA slots that could not find appropriate NOP slots have to be included in the VLIW packets exclusively created for them. Thus, the experimental result shows the actual code size reduction by less than 50%, but we still have a substantial reduction since a large number of existing NOP slots were able to turn into ROA slots. In order to find how many ROA slots reuse the space for NOP slots, we counted the numbers of NOP and ROA slots respectively for each processor. Figure 2.20 shows the sum of NOP and ROA slots in the 16VLIWP code normalized to the sum of those in the 32VLIWP code. Each bar in the graph is composed of three segments t , m and b . The middle segment m represents the number of ROA slots that have replaced the existing NOP

slots during the scheduling, and the bottom one b does the number of NOP slots that still remain in the 16VLIWP code after the conversion is complete. Clearly $m + b$ corresponds to the normalized total number ($= 1$) of NOP slots that initially exist in the 32VLIWP code. The top segment t represents the number of ROA slots that have been stored in the newly created packets. The sum $t + m$ is, therefore, the total number of ROA slots residing in the 16VLIWP code. The ratio, $m / (t + m)$, evaluates 0.66 on average for the benchmarks. This means that about 70% of the ROA slots utilize existing NOP slots. Thus, significantly more than half of the ROA slots do not induce the code size increase due to the ROA insertion, which permits us the overall code size reduction close to 50% in 16VLIWP.

The code size is also reduced in the VLES processors as shown in Figure 2.19. The average rate of the code size reduction from 32VLESP to 16VLESP is about 27%, which is obviously smaller than that of the code size reduction in the VLIW processors since there does not exist NOPs in the VLES code as stated in Section 2.4. However, since the amount of code size reduction due to the instruction bit-width reduction is bigger than that of code size increase due to the ROA slots, we can achieve substantial reduction of code size close to 30% in 16VLESP. Interestingly, there is little difference in terms of the code size between 16VLIWP and 32VLESP. In other words, both 16VLIWP and 32VLESP represent similar level of code size reduction compared to the code size of 32VLIWP. The code size reduction in 16VLIWP is due to the ISA conversion and that in 32VLESP is due to the VLES architecture. Therefore, we cannot say that our proposed ISA conversion is superior to the VLES architecture with the perspective of the code size. However, since the instruction bit-width reduction in the ISA conversion makes less energy consumed in processors as will be explained soon, our proposed ISA conversion is more effective than the VLES architecture when considering the code size and the energy consumption simultaneously.

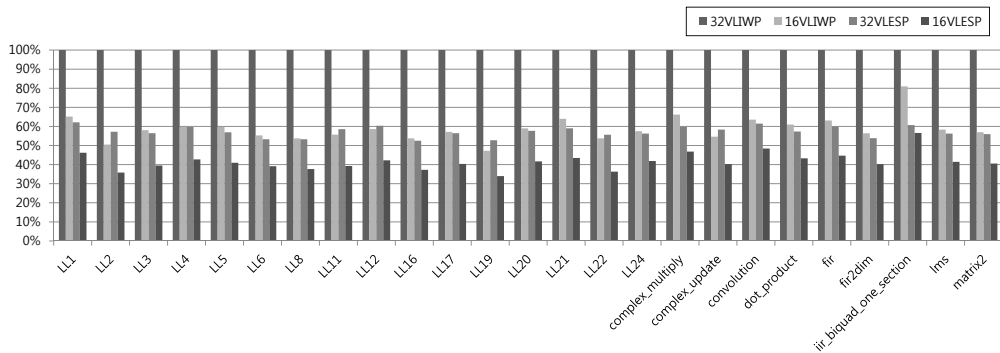


Figure 2.19 Code size of each processor (normalized to 32VLIWP)

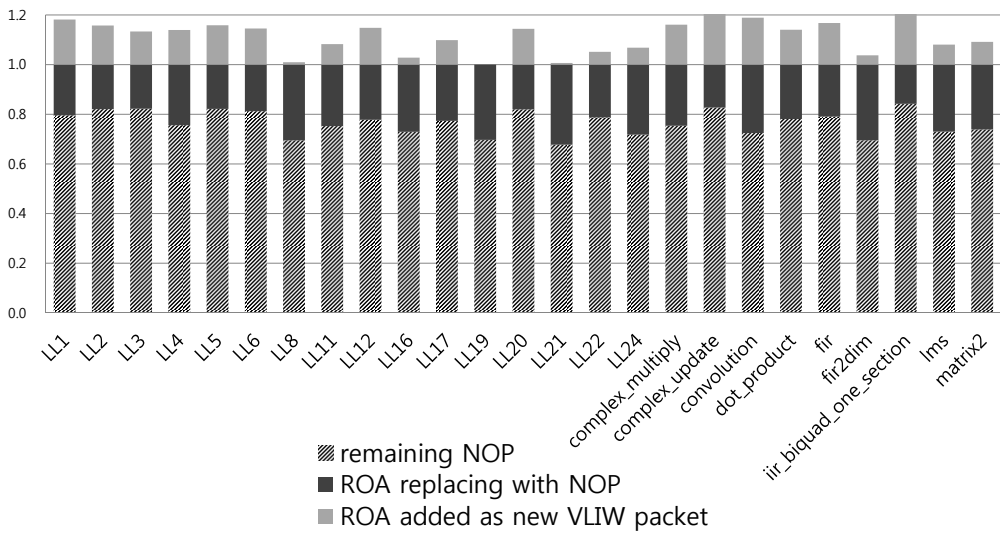
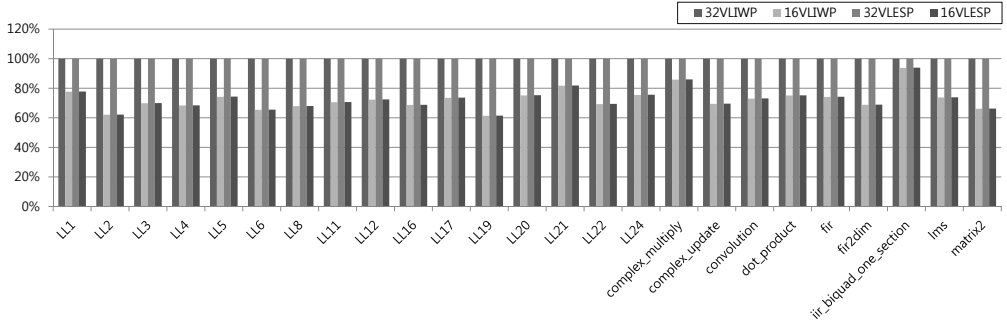
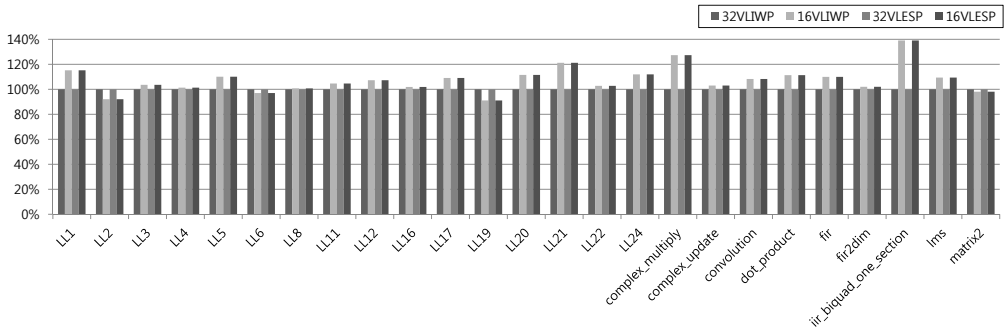


Figure 2.20 Utilization of NOP slots for ROA slots



(a) Fetch energy consumption



(b) Fetch count

Figure 2.21 Fetch energy consumption and fetch count of each processor (normalized to 32VLIWP)

Energy and Execution time

In addition to the code size, energy consumption that is another key concern in embedded systems is also estimated in our experiments. To estimate energy consumed by the instruction cache for fetching VLIW packet, CACTI 6.5 [44] was used. Figure 2.21(a) shows the fetch energy consumption of each processor. The fetch energy consumption of 16VLIWP is decreased by about 28% compared to that of 32VLIWP. This energy saving comes from the instruction bit-width reduction of our proposed ISA conversion. In fact, the VLIW packet fetch count of 16VLIWP is increased by about 7.6% compared to that of 32VLIWP as shown in Figure 2.21(b). The ROA slots requiring new VLIW packets cause this slight fetch count increase, which might result in more

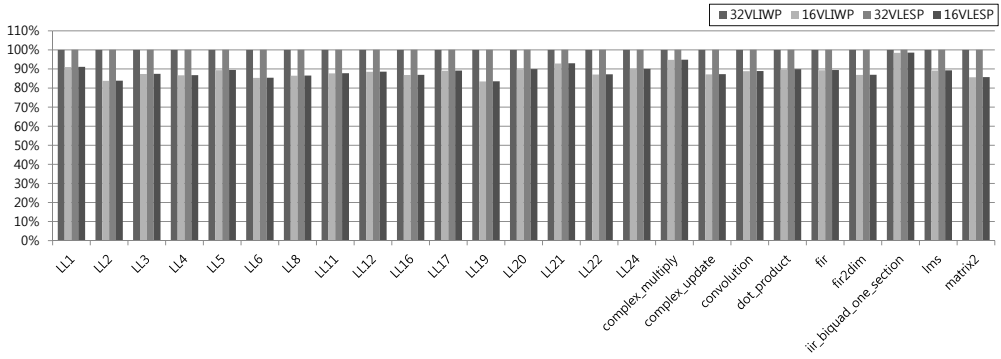


Figure 2.22 Total energy consumption of each processor (normalized to 32VLIWP)

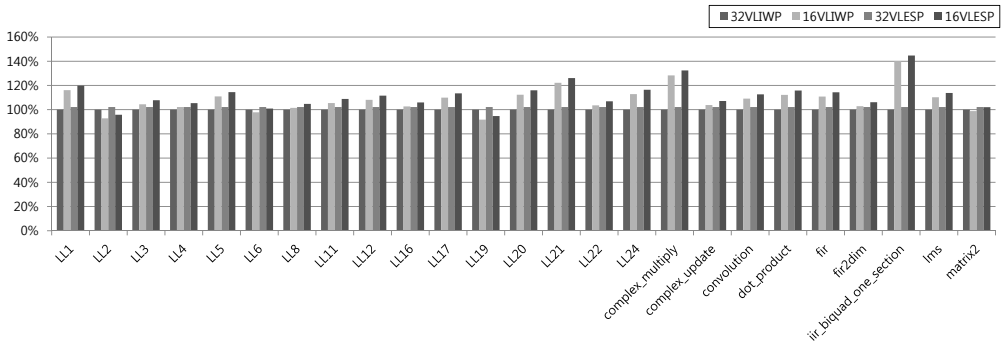


Figure 2.23 Execution time of each processor (normalized to 32VLIWP)

fetch energy consumption that is roughly proportional to the fetch count. However, when each VLIW packet is fetched, the consumed energy for the fetch in 16VLIWP is reduced by about 34% compared to that in 32VLIWP according to CACTI model [44]. As a result, even when the fetch count increases, the total fetch energy consumption in 16VLIWP is cut down to approximately 72% of that in 32VLIWP. In other words, the reduction rate of total fetch energy consumption in 16VLIWP is 28%. Since the instruction fetch energy can occupy up to 30%~45% of the total energy consumption for modern embedded processors [45], we can save the total energy consumption in the system approximately 8%~16% through the large reduction in the total fetch energy consumption as shown in Figure 2.22, representing the total energy consumption of

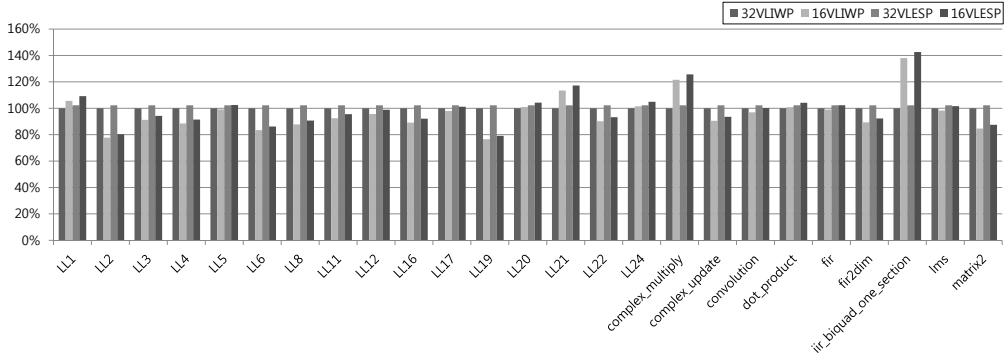


Figure 2.24 Energy-delay product of each processor (normalized to 32VLIWP)

each processor. To estimate the total energy consumption, we have also considered the additionally consumed energy for the ROA buffer access as well as the reduced fetch energy. From the experiments, the energy consumption overhead caused by accessing the ROA buffer is about 3% of the fetch energy consumption, which has negligible effect on the total energy consumption (0.5%~1% overhead).

In the case of the VLES processors, the fetch energy consumption of 16VLESP is also decreased by about 28% compared to that of 32VLESP for the same reason in the VLIW processors. Actually, in our experiments, the fetch count of each VLES processor is identical to that of the VLIW processor with the same instruction bit-width (i.e., the fetch count of 32VLESP equals to that of 32VLIWP, and the fetch count of 16VLESP is also the same as that of 16VLIWP). From this, we can see that a VLES architecture does not affect runtime behavior since the removal of NOPs in the VLES architecture cannot prevent functional units from being idle at run time as mentioned in Section 2.4. Therefore, the effect of our proposed ISA conversion on the VLES processors is the same as that on the VLIW processors with the perspective of runtime behavior, which gives us the identical results in terms of the fetch count and the fetch energy consumption in both VLIW and VLES processors.

Instead of achieving code size reduction and less energy consumption, the exe-

cution times in 16VLIWP and 16VLESP have been increased compared to those in 32VLIWP and 32VLESP, respectively, as shown in Figure 2.23. The execution time T is calculated by the formula [46].

$$T = (\text{Dynamic Instruction Count} * CPI) * (\text{Clock cycle time}) \quad (2.2)$$

The definition of CPI is the clock cycles per instruction. In our experiments, we assume $CPI \approx 1$, interpreting an instruction in CPI as a single VLIW packet. The dynamic instruction count is equal to the fetch count which is represented in Figure 2.21(b). The clock cycle time has already been measured in Table 2.3. From the formula and the meaning of each term, we can see that more fetch counts and longer clock cycle times in 16VLIWP and 16VLESP induce a slight increase of the execution time, which is approximately 8% on average over benchmarks. However, in certain benchmarks, 20%~40% increase of execution time occurred. This is because some kinds of applications have a high level of parallelism, thus generating extra VLIW packets for ROA slots which cause non-negligible amount of execution time increase. From this, as will be explained in the next subsection, we note that the effectiveness of our proposed ISA conversion heavily depends on the degree of parallelism in applications. This is also identified from the energy-delay product as shown in Figure 2.24. The energy-delay product varies from 76% to 138% according to the inherent parallelism of applications. Therefore, we need to consider the level of parallelism of applications to apply the ISA conversion and further we also need to improve the ISA conversion to overcome the limitation due to the ILP, which will be studied in our future work.

In summary, by converting the existing 32-bit ISA into our 16-bit one in both VLIW and VLES processors, we conclude that we have been able to reduce code size and energy consumption considerably at the expense of longer execution time.

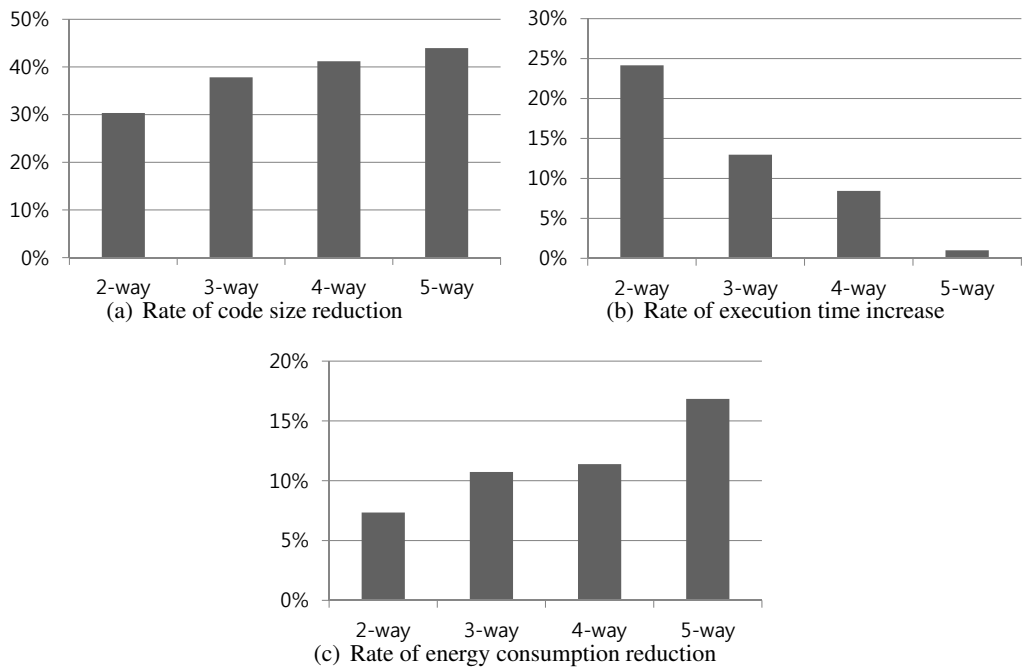


Figure 2.25 Results of converting 32-bit ISA to 16-bit ISA for each n -way VLIW processor ($n = 2, 3, 4, 5$)

2.5.3 Sensitivity Analysis

The effectiveness of the proposed ISA conversion depends on various design parameters in VLIW architectures, such as the instruction bit-width, the number of instructions and the number of issue slots, etc. Among these parameters, the number of issue slots is the most related one to the effect of the ISA conversion since it highly affects the number of NOP slots, which in turn decides the amount of overhead in terms of code size and execution time caused by the ROA slots. We applied our ISA conversion to VLIW processors with different number of issue slots. Figure 2.25 shows the results of the ISA conversion in terms of code size, execution time and energy consumption according to the change of the number of issue slots. Figure 2.25(a) shows that the rate of the code size reduction is increased as the number of issue slots increases. This is because we can have more NOP slots for accommodating the ROA slots in the code when the VLIW architecture supports more issue slots. We observe that, in case of 2-way VLIW, it still achieves substantial code size reduction close to 30%. This is not surprising since we already have 27% code size reduction for the VLES architecture where there are no existing NOP slots, implying that the lower bound of the code size reduction by the ISA conversion is about 27%. Unlike the code size, the rate of the execution time increase is increased as the number of issue slots decreases since we need more additional cycles to execute extra VLIW packets generated due to the lack of NOP slots in VLIW architectures with less number of issue slots (See Figure 2.25(b)). Figure 2.25(c) shows that we can achieve less energy reduction as the number of issue slots decreases. This is also because more energy is consumed to execute extra VLIW packets generated due to the lack of NOP slots in less number of issue slots.

Our proposed ISA conversion is also affected by the characteristics of applications as well as VLIW design parameters. For example, if the ISA conversion is applied to applications with high ILP (instruction-level parallelism), the results in terms of

code size, execution time and energy consumption would be worse than the case of applications with low ILP. Therefore, characterizing applications, especially focusing on ILP, is important for applying our ISA conversion. The impact of ILP is intuitively grasped from the results about the influences of the number of issue slots in VLIW architectures as explained above. The high ILP causes the lack of NOP slots and thus it will show a similar tendency with the case of less number of issue slots that also suffers from the lack of NOP slots. On the contrary, the low ILP corresponds to more number of issue slots in that both have relatively enough NOP slots.

Through this sensitivity analysis, we note that our proposed ISA conversion would be less effective when it is applied to VLIW architectures with less number of issue slots or to applications with higher ILP. In our future work, we will strive to tackle the difficulty of lack of NOP slots in less number of issue slots or higher ILP to extend an applicable range of our ISA conversion.

2.6 Related Work

The purpose of our work is not only to reduce code size but also to reduce the instruction word length for low cost and low power hardware implementation. In this sense, our work differs from many earlier approaches that aim to reduce only code size. One category of such approaches is the compiler research that attempts to alleviate the code bloating problem by making VLIW code denser via more aggressive, inter-block scheduling, such as trace scheduling, superblock scheduling and hyperblock scheduling [35, 36, 37]. In fact, these techniques are somehow complementary to our work because they can be additionally applied to our compilation framework so as to improve the code quality of our 16-bit ISA VLIW processor. Another interesting approach might be the work [45] that tries to reduce code size by packing instructions. It proposes the *instruction register file* (IRF) to achieve code reduction. The compiler selects up to 32 instructions and stores them in the IRF. The instructions can be refer-

enced via an index to each one. Besides normal instructions, the machine also provides *packed* instructions that can reference up to five instructions from the IRF. H.Lin, et al [47] extended the IRF work from single-issue architecture to VLIW architecture. Commonly occurring instructions are first selected and stored in the IRF for fast access in program execution. Synchronization among VLIW issue slots is ensured by introducing new instruction formats and micro-architectural support. However, the IRF-based approach requires profiling information to explore frequently executed instructions for reducing the code size.

Existing NOP compression techniques [48] [49] are irrelevant to the instruction bus width reduction. They also focus only on the code size reduction. If a NOP compression technique is applied to a 32-bit ISA hardware alone without our technique, code size could be reduced but the instruction bus width does not change since it has still 32-bit ISA. So power consumption reduction is only achieved from the code size reduction. However, if both of our technique and a NOP compression technique are applied simultaneously, more power consumption reduction could be achieved due to the code size reduction and the reduced instruction bus width. Similarly, instruction compression techniques [50] [51] also do not consider the instruction bus width reduction. Since these techniques are orthogonal to our technique, they are combined to our technique in order to get more code size reduction.

Regarding power consumption, there is a noticeable technique, called *dictionary code compression* [52], which reduces the power consumed in the instruction fetch path of processors. Instructions or instruction sequences in the code are replaced with short code words. These code words are later used to index a dictionary that contains the original uncompressed instruction or an entire sequence. However, this technique requires complex decoder logic for encoded instructions. Although we also need additional decoder logic for ROA manipulation, our technique does not need pre-profiling that is indispensable for them to determine which instructions are stored in the dic-

tionary. Moreover, their technique does not directly help to decrease bus-bandwidth requirements since it does not intend to reduce instruction word length itself.

Lastly it would be worthwhile to consider native 16-bit processors. As explained earlier, 16 bits are not long enough to provide a sufficient word length for a regular encoding scheme to include all necessary instructions in the instruction set. Therefore, 16-bit ISAs are hardly employed except by ultimately low-end systems, for which extremely small size and low power consumption are of high priority. A plausible way to overcome this limitation is to aggressively apply a vertical encoding scheme with special addressing modes, such as *dedicated* and *implied*, to their instructions, where each instruction is to specify as its operands a confined subset of the hardware registers available in the processor. To support dedicated addressing, the machine distributes its registers physically into several register files each dedicated to one of its functional units. For example, D950 [53] has seven data registers in total. The registers are physically distributed into four register files: left registers **L0/L1**, right registers **R0/R1**, accumulators **A0/A1** and a multiplier register **P**. Its multiply instruction has three operands with dedicated addressing modes, where four registers are bound to one source operand, and the register **P** is to the destination operand. So we need only two bits to specify the source, and for the destination, we do not even need to allocate any bit as we use an implied addressing mode. Such heterogeneity of register architecture (HRA) normally leads the processor to have a *non-orthogonal* instruction set because the use of each register file is differentiated for individual instructions in the code. Unfortunately, code generation for non-orthogonal ISAs ask for far more complex algorithms [41, 54] than conventional ones that have been developed for orthogonal RISC-style ISAs. This is because when the compiler selects an instruction, it needs to simultaneously determine among multiple register files which register is to be allocated for each operand of an instruction. This problem is notoriously known as *phase coupling*. To recap, the strategy of exploiting dedicated addressing may reduce

the width of operands, but it triggers the phase coupling problem that will prohibitively complicate compiler code generation.

As for VLIW architectures, we have found that virtually no commercial machine adopts a 16-bit ISA alone even if their target is a low-end system. For example, Carmel [55], an Infineon VLIW processor, has 16-bit data paths in order to attain low power consumption and low system cost, yet it supports 24-bit and 48-bit instructions only. This is mainly because VLIW machines strive to gain performance boost via a horizontal encoding scheme based on a regular RISC-style ISA [56], being reluctant to include the aforementioned vertical addressing modes that may result in non-orthogonal instruction sets. On the other hand, our VLIW machine executes basically any 32-bit instructions inside the CPU while it stores and fetches instructions in a 16-bit instruction format. This implies that the machine can maintain the same orthogonal register architecture and instruction set as the original 32-bit machine, consequently enabling the compiler to use conventional code generation algorithms.

Chapter 3

Compiler-assisted Dynamic Code Duplication Scheme for Soft Error Resilient VLIW Architectures

In order to minimize the code size and to effectively increase the reliability with least overheads for duplicating instructions, we propose a novel approach, *compiler-assisted dynamic code duplication method* and *vulnerability-aware duplication algorithms* for VLIW processors. The proposed VLIW architecture accepts an assembly code composed of only original instructions as an input, and generates duplicated instructions at run-time with the help of encoded information attached to original instructions. When the compiler generates the assembly code, it is determined whether an original instruction will be duplicated or not at run-time, and then the result of the decision is included in the encoding space of the original instruction. Since the duplicates of original instructions are not explicitly present in the assembly code, the increase of code size due to the duplicated instructions can be avoided in the proposed technique. Also, the compiler-assisted duplication algorithms provide mechanisms considering vulnerabil-

ity of each instruction so that our approach can offer selective protection under the limited budget of power and performance so they can provide higher reliability than the previously proposed techniques unaware of different vulnerability levels of instructions. The vulnerability-aware duplication algorithms take into account two metrics: (i) *temporal vulnerability* based on the more often executed, the more vulnerable, and (ii) *physical vulnerability* based on the larger cell area (more number of transistors), the more vulnerable.

The contributions and results of this work include:

- we present a compiler-assisted dynamic code duplication scheme for VLIW architectures which can reduce the code size significantly.
- we propose vulnerability-aware duplication algorithms which can improve the reliability effectively with minimal costs.
- The experimental results show that the proposed VLIW architecture is implemented with 3.2% area overhead and no clock cycle penalty as compared to an existing technique.
- The experimental results demonstrate that our proposals can reduce the code size by up to 40% and detect more soft errors by up to 82% via fault injection experiments over a suite of benchmarks as compared to the previously proposed technique.

3.1 Related Work

With technology scaling, soft errors are becoming an important design concern in embedded systems. Soft errors have already been revealed to cause fiscal damages [24]. For example, Sun blamed soft errors for the crash of their million-dollar line SUN flagship servers in Nov.2000 [25]. In one incident, soft errors crashed an interleaved

system farm. In another incident, soft errors brought a billion-dollar automotive factory to halt every month [57]. Further, highly integrated chip equipped reliability-sensitive embedded devices such as mobile health-care systems and anti-lock braking systems (ABS) in automotive engine control units (ECU) are significantly threatened by exponentially increasing soft error rates with technology scaled. Thus, it is a necessity to combat soft errors for embedded systems in both emerging and traditional computing environments.

Previous works for coping with soft errors have been based on redundancy. Redundancy has been applied at different levels of granularity, such as hardware level, thread level, and instruction level, etc. Techniques for exploiting n-modular redundancy (nMR) [58] check soft errors with redundant hardware components and thus incur high overheads in terms of area and power consumption [59, 60]. The appearance of simultaneous multithreading (SMT) capabilities in modern processors gives an opportunity for soft error detection by running two copies of one thread and comparing their outcome [61, 62, 63]. The drawbacks of these approaches include substantial performance degradation, hardware cost, and power consumption increase.

Several researches have investigated redundancy techniques at the instruction level for soft error detection. Unlike aforementioned techniques highly dependent on hardware features, they achieve redundant execution by relying on software techniques, with little or no hardware cost. As one of promising compiler-based software approaches for soft error detection, SWIFT [64] duplicates program's instructions, schedules the original and duplicated instruction sequences together in the same thread of control, and inserts explicit validation codes to compare the results from the original instructions and their corresponding duplicates. CRAFT [65] and PROFIT [65] enhance SWIFT approach by leveraging extra hardware structures and applying partial protection based on AVF (Architectural Vulnerability Factor) analysis [66], respectively. These approaches provide complete fault coverage with minimal area cost. However,

they incur significant performance overhead since the number of instructions can be easily doubled mainly due to full duplication of instructions.

In the context of redundancy at the instruction level, Jie Hu et al. [18, 11] propose techniques to mitigate the impact of soft errors on reliability by duplicating instructions in VLIW architectures, which are of our interest. The main idea behind their approach is to fill empty slots (NOP instructions) with duplicated instructions without performance penalty if there exist available empty slots. Otherwise, it copies duplicated instructions at new instruction cycles. As compared to the previous instruction duplication studies [64, 65], this approach can also increase the reliability since it can detect soft errors by comparing the output of an original instruction with that of the duplicated instruction. Interestingly, this approach can improve the reliability under constraints of power consumption, performance, and code size by static analysis at compile-time. It can trade off reliability at the cost of performance by adjusting the rate of duplicate instructions as opposed to full duplication of instructions in the previous works [64, 65]. Therefore, this approach can be exploited in various forms of application requirements from reliability-sensitive to performance-sensitive ones. Although this approach called *static code duplication scheme* in this dissertation is very promising in the area of the instruction-level redundancy, it has two primary drawbacks: (i) *the increase of code size* and (ii) *unawareness of different importance among instructions*. In contrast, our approach does not incur the code size overhead and considers different levels of vulnerability in instructions when duplicating instructions.

Recently, several selective protections have been proposed to increase the reliability. S. Rehman et al.[67] propose reliability-aware code transformation techniques to duplicate instructions under the performance constraint. Their techniques are very promising to increase the reliability with the least performance overhead by transforming the codes while our approach presents a dynamic code duplication to reduce the code size and to consider instruction vulnerability for duplication. Note that their tech-

niques are orthogonal to our dynamic code duplication and thus theirs can be applied for our proposed architectures. N. Nakka et al. [68] present processor level selective replication instead of entire replication. This methodology can improve the performance because their compiler can ignore benign errors. Their scheme is exploited at the processor level selective replication while ours is at the instruction level selective duplication. D. Borodin et al. [69] propose the efficient instruction duplication scheme by exploiting precomputation and memoization. It improves the performance and fault coverage of permanent fault. However, their technique deals with the permanent faults while our technique is presented for the transient faults such as soft errors in VLIW architectures. There have also been several researches for out-of-order processors to dynamically generate code against soft errors [70, 71, 72]. Among these approaches, X. Vera et al. [72] propose a selective replication scheme close to our approach with the perspective of considering the vulnerability of instructions. They protect only a subset of instructions, the most vulnerable ones, by selectively replicating those with higher vulnerabilities than a predefined vulnerability threshold. In their scheme, vulnerability of instructions is estimated based on the cell area they occupy and the time they spend in the issue queue, a part of the dynamic scheduler for out-of-order execution. In other words, this scheme exploits a dynamic scheduler for out-of-order processors to duplicate and schedule instructions. However, this approach has also two drawbacks: (i) *significant hardware cost due to the dynamic scheduler* and (ii) *inflexibility as a hardware-based approach*.

In contrast, our approach incurs the least area overhead since ours exploits VLIW architectures which do not need the dynamic scheduler. Further, as a software-based approach, our scheme has advantages in that ours can exploit global information available at compile-time such as a loop structure and adjust the threshold value at compile-time without redesigning the hardware architecture. Also, our method proposes methodology to combine physical or spatial vulnerability and temporal vulnerability by off-

line profiling. Therefore, our approach can improve reliability from various angles. In our experiments, we will present the effectiveness of our software-based scheme as compared to the hardware-based selective replication scheme.

3.2 Compiler-assisted Dynamic Code Duplication

We propose a *compiler-assisted dynamic code duplication scheme* for VLIW architectures. Our purpose of instruction duplication is to mitigate soft error impacts on datapath, in particular, ALU (IALU and FALU) and LSU (Load/Store Unit). All the other components are assumed to be protected in an appropriate way. For instance, instruction cache, data cache, and general purpose register files can be protected with parity. Also we assume that buses, queues, comparators, and other registers are protected as well. For our proposed scheme, our compiler can generate scheduling information embedded in the code and help our modified VLIW architecture duplicate instructions at run-time rather than at compile-time. This is why our approach can resolve the issue of the increased code size in the static code duplication scheme since duplicated instructions are not explicitly present in our code before run-time while they are present in the code of the static code duplication scheme.

We have implemented our VLIW architecture for dynamic code duplication scheme by modifying that of the static code duplication scheme [11]. Figure 3.1 shows the datapath of both VLIW architectures where the modified part is highlighted in the shade. Indeed, the fetch stage only needs to be modified mainly because our scheme needs to decode and use the embedded information for instruction duplication generated by our compiler. At the fetch stage, a sequence of consecutive instructions, called a *fetch packet*, is read from the program memory, and each instruction of the fetch packet is sent to the decode stage according to the functionality of each issue slot. The bundle of instructions sent to the decode stage is called an *execute packet*, which is identical to the fetch packet in the case of the static scheme in general and in the static

completes, it writes the value into the output register as well as in the RVQ. When the duplicate instruction of this instruction completes, its output is compared with the content of the entry in the RVQ associated with the original instruction. Therefore, our dynamic code duplication scheme also eliminates the need of checking instructions for validation by taking advantage of IRVQ/FRVQ and LSAQ since our dynamic scheme also exploits these features. In both the static scheme and our dynamic scheme, an original instruction and its duplicate one identically behave except that RVQ and LSAQ are only written by the original. Other hardware components, such as arithmetic logic unit, address generation unit, bypassing unit, etc., are equivalently exploited for both original and duplicate instructions. One thing we need to keep in mind is that the identical parts between two schemes are the *pipeline architectures* after the fetch stages, not the *execution behaviors*, since we only modified the fetch stage while maintaining the other stages (DC, EX, MEM, and WB stages) unchanged from the static scheme. Even though the pipeline stages except for the fetch stages are identical, the execution behaviors of two schemes should not be the same since the code generations of two schemes are different from each other. In the following subsections, we will describe our dynamic code duplication mechanism and its modified fetch stage in more detail.

3.2.1 ISA Design

When a fetch packet is converted to an execute packet at the fetch stage, configuration information for the execute packet should be given in a certain mechanism. To minimize the hardware overhead for duplicate instructions, our dynamic scheme considers three possible duplication cases in addition to no duplication. For this, we designate two bits, $D0$ and $D1$, in each instruction as summarized in Table 3.1. They indicate whether an original instruction is duplicated or not at run-time. They also indicate whether it is scheduled at the current cycle or at the next cycle if it is duplicated. Further, they indicate whether in a new packet or not if it is at the next cycle. First, a

cycle	slot 0	slot 1	slot 2	slot 3
t	A	NOP	B	C
t	A	A'	B	C

(a) Duplicated to replace a NOP at the same cycle: (D0,D1)=(0,1)

cycle	slot 0	slot 1	slot 2	slot 3
t	A	B	NOP	NOP
t+1	NOP	NOP	C	D
t	A	B	NOP	NOP
t+1	A'	B'	C	D

(b) Duplicated to replace a NOP at the next cycle: (D0,D1)=(1,0)

cycle	slot 0	slot 1	slot 2	slot 3
t	A	B	C	D
t+1	E	F	G	H
t	A	B	C	D
t+1	A'	B'	C'	D'
t+2	E	F	G	H

slot 0, 1 : ALU instructions
slot 2 : multiply, floating point operations
slot 3 : memory access instructions (load, store)

(c) Duplicated within a new VLIW packet at the next cycle: (D0,D1)=(1,1)

Figure 3.2 Three possible cases for generating instruction duplication (A shade one represents a duplicate instruction, (D0,D1)=(0,0) means no duplication.)

duplicate is generated at the same cycle as its original instruction. As shown in Figure 3.2(a), the duplicate of an instruction *A* at slot 0 is generated at slot 1 where there is a NOP available. Second, a duplicate is generated to replace a NOP at the next cycle. Figure 3.2(b) shows that the duplicate of *A* cannot be generated at cycle *t* since there is not a NOP but *B* at slot 1. Even though there are NOPs at slot 2 and slot 3, the duplicate of *A* cannot be executed at those slots due to the constraints of issue slots. In this case, if there is a NOP at the same issue slot of the next cycle, a duplicate can be generated to replace the NOP. Otherwise, a new VLIW packet should be generated for duplicates, as shown in Figure 3.2(c). This is the third case.

Note that it is definitely not hard to find out two bits unused space in general in 32-bit instruction set architecture [73] and D0 and D1 can be assigned into existing

Table 3.1 The meaning of D0 and D1 to duplicate instructions dynamically

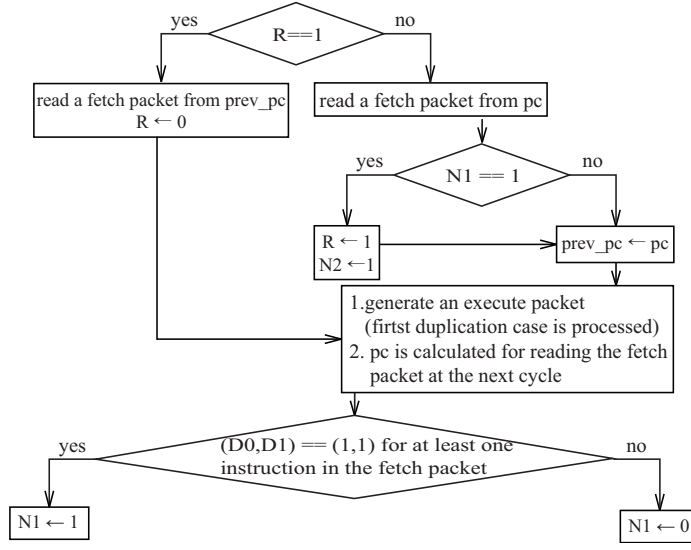
D0	D1	Implication
0	0	Not Duplicated
0	1	Duplicated to replace a NOP at the same cycle
1	0	Duplicated to replace a NOP at the next cycle
1	1	Duplicated within a new VLIW packet at the next cycle

encoding space of instructions without overheads of space and loss of instructions [11].

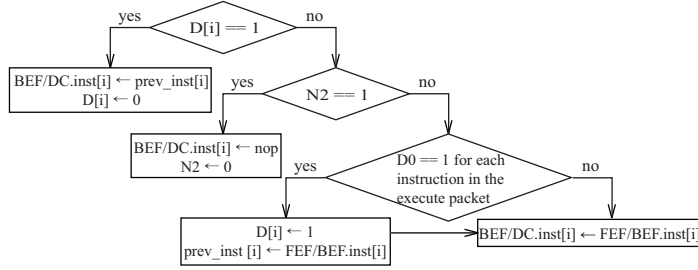
To distinguish each case at run-time, the configuration information should be embedded in the instruction encoding space at compile-time. Otherwise, the hardware cannot avoid being more complicated because it should dynamically check data dependencies among instructions. Also, we consider scheduling the duplicate instruction by the next cycle, not further, to minimize the complexity overhead of the hardware implementation. Thus, the duplication range, the range of cycles where a duplicate instruction can be scheduled, is two cycles in our scheme. However, this limitation of the duplication range does not incur performance overhead in our experimental results as will be presented in Section 3.4.2. Note that our compiler-assisted dynamic code duplication scheme is orthogonal to further complicated VLIW architectures and there should be interesting tradeoff space between the complexity and the performance, which is definitely a topic for our future work.

3.2.2 Modified Fetch Stage

Our approach needs to separate the fetch stage into two stages, FEF and BEF. It could have increased clock cycle time if decoding $D0$ and $D1$ would be merged into the fetch stage or into the decode stage, which would have made a negative impact on the performance at that stage. Figure 3.3 depicts behavior flow charts for two separate FEF and BEF stages. Global registers such as R , $prev_pc$, $N1$, and $N2$, and local registers used for each issue slot such as $D[i]$ and $prev_inst[i]$ are newly introduced. The global registers are for the third case. R notifies that the fetch packet of the next cycle should



(a) Front-End Fetch (FEF) Stage



(b) Back-End Fetch (BEF) Stage

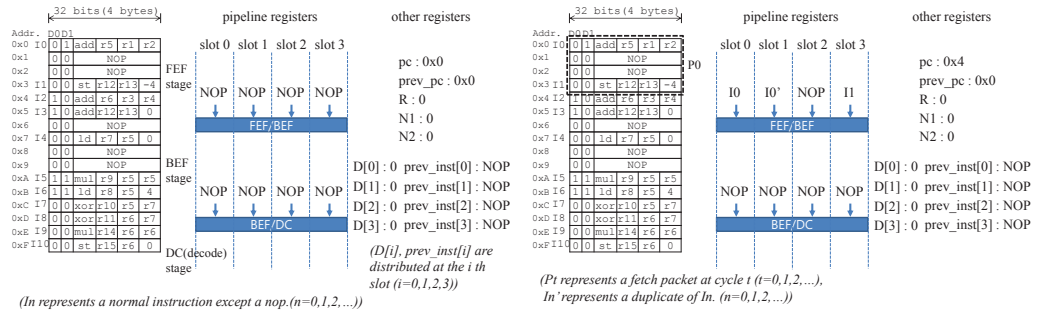
Figure 3.3 Behavior flow chart of modified fetch stage

be the same as that of the current cycle. For this, *prev_pc* stores the address of the fetch packet of the current cycle. *N1* indicates that the current fetch packet at FEF stage requires a new VLIW packet for its duplicate, and *N2* represents whether the current execute packet at BEF stage is an original or a duplicate. The local registers are for the second and the third cases. *prev_inst[i]* stores the original instruction and delivers it to the decode stage by writing it into BEF/DC pipeline register if *D[i]* is set to 1, which means the duplication of this instruction. Using these registers, FEF and

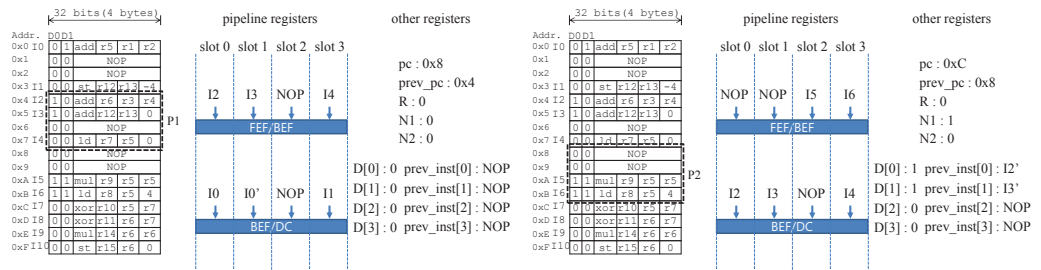
BEF stages work in a combined manner to support dynamic instruction duplication. As shown in Figure 3.3(a), FEF stage behaves differently according to the values of R and $N1$, and processes the first case, i.e., $(D0, D1) = (0, 1)$. BEF stage also processes the second and third cases according to the values of $D[i]$, $N2$ and $(D0, D1)$ of each instruction in the execute packet as shown in Figure 3.3(b).

To help understand our scheme, Figure 3.4 presents an example how FEF and BEF stages work for each case of generating duplicated instructions. Figure 3.4(a) shows an example code and initial states of pipeline registers and other registers. Figure 3.4(b) shows the architectural state after execution at cycle 0. $I0'$, a duplicate of $I0$, is generated at FEF stage and is placed in slot 1 since $(D0, D1)$ of $I0$ equals $(0, 1)$. Thus, Figure 3.4(b) shows the first case. Next, Figure 3.4(c) shows the second case since $(D0, D1)$ equals to $(1, 0)$, respectively, in each of $I2$ and $I3$ in $P1$. Duplicated instructions, $I2'$ and $I3'$, are stored in $prev_inst[0]$ and $prev_inst[1]$, respectively, at cycle 2. Also, both $D[0]$ and $D[1]$ are set to 1. Then, BEF stage delivers $I2'$ and $I3'$ to the next pipeline stage at cycle 3 since both $D[0]$ and $D[1]$ are equal to 1 (See Figure 3.3(b) and Figure 3.4(e)). Figure 3.4(e) shows that $I2'$ and $I3'$ are placed in the NOP slots, slot0 and slot1, of $P2$. Thus, $I2'$, $I3'$, $I5$, and $I6$ are executed at the same cycle. Note that we need avoid data dependency violation among $I2'$, $I3'$, $I5$, and $I6$, which is guaranteed with the help of instruction duplication algorithm at compile-time.

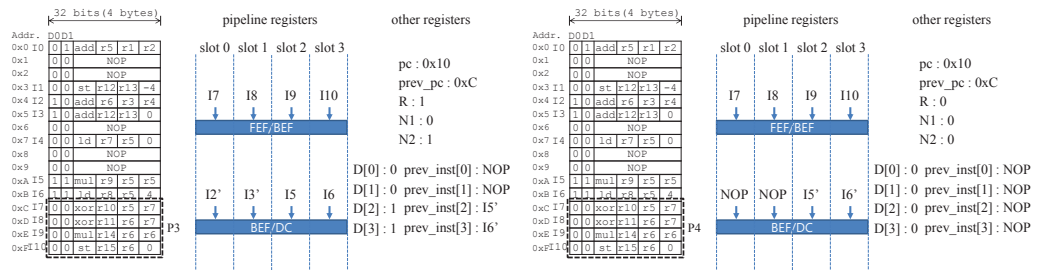
The last case for duplicating instructions is the third, where both $D0$ and $D1$ are equal to 1. Duplicated instructions, $I5'$ and $I6'$, are stored in $prev_inst[2]$ and $prev_inst[3]$, respectively, at cycle 3 as shown in Figure 3.4(e). Also, both $D[2]$ and $D[3]$ are set to 1. In this case, $P3$ does not have NOPs that could be replaced with duplicated instructions, so a new VLIW packet is generated for the duplicate of $P2$ and then the new packet is delivered to the decode stage at cycle 4 as shown in Figure 3.4(f). This new packet generation is processed at BEF stage as the behavior represented in Figure 3.3(b). Since $D[2]$ and $D[3]$ are equal to 1, $I5'$ and $I6'$ are delivered



(a) an example code : program memory and an initial state of the fetch part of pipeline architecture (b) the state of the pipeline architecture at the end of cycle 0



(c) the state of the pipeline architecture at the end of cycle 1 (d) the state of the pipeline architecture at the end of cycle 2



(e) the state of the pipeline architecture at the end of cycle 3 (f) the state of the pipeline architecture at the end of cycle 4

Figure 3.4 An example: Behaviors of modified fetch stage according to each case

to the next pipeline stage at cycle 4 as shown in Figure 3.4(f). In the case of slot 0 and slot 1, NOPs should be delivered to the next pipeline stage at cycle 4. For this, $N2$ is set to 1 at cycle 3 in Figure 3.4(e). Otherwise, $I7$ and $I8$ could be delivered to the next pipeline stage with $I5'$ and $I6'$, which incurs an incorrect result of the program. Due to the generation of the new packet, $P3$ should be delivered to the decode stage at cycle 5 instead of cycle 4. For this, FEF stage at cycle 4 needs to read the same fetch packet at cycle 3, i.e., $P4$ should be identical to $P3$. Otherwise, $P3$ cannot be delivered to the decode stage, and program would not work properly. To read the same fetch packet in consecutive two cycles as shown in Figure 3.4(e) and Figure 3.4(f), R and $N1$ are used as flags. At cycle 2, our architecture understands the necessity of a new packet to generate the duplicate of $P2$ since $(D0, D1)$ are equal to $(1, 1)$ for $I5$ and $I6$. Thus, $N1$ is set to 1 at cycle 2 (See Figure 3.4(d)) and therefore R is also set to 1 at cycle 3 (See Figure 3.4(e)). Since R is 1 at the start of cycle 4, a fetch packet is read from $prev_pc$ (See Figure 3.3(a)) and it becomes possible to read the same fetch packet at both cycles 3 and 4 (See Figure 3.4(f)).

In this section, we describe our architecture based on previously proposed VLIW architecture for compiler-assisted dynamic code duplication. The following will present our compilation and instruction duplication techniques for our VLIW architecture.

3.3 Compilation Techniques

Our proposal resolves two issues: (i) code size reduction and (ii) vulnerability-aware instruction duplication.

Our compiler-assisted dynamic code duplication scheme is able to reduce the code size by dynamically duplicating instructions in the modified fetch stage in VLIW architectures as described in the previous section and to effectively increase reliability by taking the different degrees of instruction vulnerabilities into account in duplicating instructions. In the following subsections, we will talk about the previously proposed

static code duplication scheme and our dynamic code duplication scheme which is aware of vulnerability to determine which instructions to be duplicated at compile-time.

3.3.1 Static Code Duplication Algorithm

As stated in Section 3.1, the static code duplication scheme [11] can trade off performance loss with required degree of reliability by adjusting the amount of duplicated instructions while other previous works fully duplicate instructions at the expense of maximum performance loss. Their approach allocates duplicated instructions if NOP is available or increases the schedule length for duplicating instructions within a duplication range which is determined by the level of allowable performance degradation.

However, their code duplication algorithm duplicates instructions in a sequential manner and its fault tolerant coverage is limited into the earlier examined instructions, especially if the power or performance constraints are limited. For instance, their duplication algorithm under no performance overhead duplicates 17 instructions out of duplicable 70 ones and they are all located within the first half in the scheduled code for benchmark *complex_multiply* when we implement their duplication algorithm and run a simple experiment. This unbalanced duplication of instructions is effective enough to increase the reliability under the performance bound if each instruction has equal impact on the reliability to others. However, several researchers [64, 74, 75, 76, 72, 77] have shown that not all data or instructions are equally important in terms of reliability. Thus, our code duplication algorithm introduces the different degrees of instruction vulnerabilities when selecting instructions for duplication to increase the reliability with minimal performance overhead, which will be described in the following subsection.

3.3.2 Vulnerability-aware Duplication Algorithm

We present three duplication algorithms considering different degrees of instruction vulnerabilities for duplicating instructions to effectively maximize the reliability.

Our first approach is *temporal-vulnerability-aware duplication (TVAD)* algorithm. Temporal vulnerability has been presented and exploited in several previous works, especially works for cache and memory protection against soft errors [78, 79]. They estimate the time period of data such as program variables in caches and protect selectively those which have higher vulnerability in terms of time above the threshold value. However, it is extremely hard to estimate which instruction executes more often than others. We suppose that estimating the vulnerability of instruction in a datapath is beyond our interest in this work and it will be definitely our future work. In our study, the first simple proposal exploits this concept of temporal vulnerability and considers instructions in the loop more important than others in terms of reliability since they have higher chance to be executed more often, which implies more chances to be exposed to soft errors. Indeed, compilation techniques suppose 10 times of execution for instructions in the loop [80] so our approach puts 10 times more vulnerability for instructions in the loop, i.e., importance in terms of reliability, than ones out of the loop when our approach selects instructions for duplication. Thus, our TVAD algorithm defines the vulnerability of instructions such that $V_I = 10 \times v$ if I is in the loop or $V_I = 1 \times v$ otherwise where V_I is the vulnerability of an instruction I and v is a vulnerability unit.

Our second approach is *physical-vulnerability-aware duplication (PVAD)* algorithm. Note that the more exposed, the more vulnerable [81, 72]. If a combinational logic consists of more number of transistors and takes up larger portion in the chipset than another logic, it is more vulnerable since it is more largely exposed to energetic particles inducing soft errors. To estimate physical vulnerabilities of instructions, we

have run a simple experiment in a compiler-simulator-synthesizer framework (see Section 4.4.1) and estimated the cell areas of instructions. Table 3.2 samples the normalized cell areas of several instructions to that of an instruction *ldc_ri*. *mul* instruction takes up more than 270 times in cell area than *ldc_ri* instruction, which can be translated into 270 times higher vulnerability of *mul* instruction than that of *ldc_ri*. Thus, it makes better sense in terms of reliability to duplicate *mul* instruction rather than *ldc_ri* if we can only select one instruction out of those two instructions due to the performance bound. Note that the critical path has been already determined from the longest delay of an instruction in the pipeline design and therefore the selection with larger cell area does not affect the performance negatively. Our PVAD algorithm defines the vulnerability of instructions such that $V_I = n_I \times V_{ldc_ri}$ where V_I is the vulnerability of an instruction I , n_I is the normalized cell area of I to that of the instruction *ldc_ri*, and V_{ldc_ri} is the vulnerability of *ldc_ri*.

Our last approach is *temporal and physical vulnerability-aware duplication (TPVAD)* algorithm combining TVAD, PVAD and basic instruction scheduling algorithm in VLIW architecture. Thus, TPVAD is vulnerability-aware duplication algorithm considering both temporal and physical vulnerability under the performance constraint. Our TPVAD can improve reliability more effectively under the constrained performance as compared to previously proposed static code duplication scheme by duplicating instructions with higher vulnerability in terms of both temporal and physical vulnerabilities. TPVAD defines the vulnerability of instructions such as $V_I =$

Table 3.2 Cell areas of instructions normalized to that of an instruction *ldc_ri*

Instruction	Area	Instruction	Area	Instruction	Area
mul	271.72	add + shift	219.40	sub + shift	213.58
xor + shift	178.06	add	170.36	or + shift	168.06
and + shift	164.41	sub	155.05	xor	129.16
or	114.83	and	109.93	store	87.03
load	69.72	cmp_rr	59.96	branch	48.77
cmp_ri	48.26	lui_ri	1.00	ldc_ri	1

TPVADuplication

Input : A VLIW code sequence without duplicate instructions in a function

Output : A VLIW code sequence with information for duplicate instructions in a function

A function has a region graph G

Let $G = (N, E)$

where $N = \{n_i \mid n_i \text{ is a region, which is normally a basic block}\}$

and $E = \{e_{ij} \mid \exists e_{ij} \text{ if there is a control flow } n_i \rightarrow n_j\}$

phy_vul_table T : a table with each instruction and its physical vulnerability

priority_list L : a list of instructions sorted in a descending order of the vulnerability

```
01: for (each  $n_i \in G$ )
02:   if ( $n_i$  is a part of loop) then
03:     temp_vul_factor = 10
04:   else
05:     temp_vul_factor = 1
06:   endif
07:   for (each instruction  $inst \in n_i$ )
08:     phy_vul =  $T.get\_phy\_vul(inst)$ 
09:     assign_vul( $inst, phy\_vul \times temp\_vul\_factor$ )
10:      $L.insert(inst)$ 
11:   endFor
12: endFor
13: for (each instruction  $inst \in L$ )
14:   dup_inst = create_duplicate( $inst$ )
15:   region = get_region( $inst$ )
16:   code_increase_margin = get_code_increase_margin(region)
17:   sched_cycle = get_sched_cycle( $inst$ )
18:   possible_cycle = sched_cycle
19:   while ( $possible\_cycle \leq sched\_cycle + 1$ ) do
20:     sched_success = try_to_schedule( $dup\_inst, possible\_cycle$ )
21:     if (sched_success == true) then
22:       break
23:     endif
24:     possible_cycle ++
25:   endWhile
26:   if (sched_success == false) && (code_increase_margin > 0) then
27:     insert_new_cycle(sched_cycle + 1)
28:     try_to_schedule( $dup\_inst, sched\_cycle + 1$ )
29:     code_increase_margin --
30:     update_code_increase_margin(region, code_increase_margin)
31:   endif
32: endFor
```

Figure 3.5 TPVADuplication– Temporal and Physical Vulnerability-aware Duplication Algorithm

$10 \times n_I \times V_{ldc_ri}$ if I is in the loop or $V_I = n_I \times V_{ldc_ri}$ otherwise where V_I is the vulnerability of an instruction I , n_I is the normalized cell area of I to that of the instruction ldc_ri , and V_{ldc_ri} is the vulnerability of ldc_ri .

Figure 3.5 describes our TPVAD algorithm. The first loop (line 01-11) makes a priority list of instructions considering temporal and physical vulnerability simultaneously. The second loop (lines 13–32) attempts to schedule duplicate instructions in priority order with perspective of its vulnerability. For each instruction in the list, L , our TPVAD duplicates the instruction, and gets the information needed for instruction duplication (lines 14–18). And it attempts to schedule the duplicate instruction within the next cycle (lines 19–25). When the schedule fails, TPVAD inserts a new cycle and schedules the instruction at the cycle if code increase margin is larger than 0. Otherwise, it gives up duplicating the instruction (line 26-31).

3.4 Experiments

3.4.1 Experimental Setup

To evaluate the effectiveness of our proposals, we have implemented a compiler-simulator-synthesizer framework as shown in Figure 4.8. Our proposed VLIW architecture has been implemented in Processor Designer of Synopsys [40]. It generates software tools such as assembler, linker, and simulator. Further, it generates HDL (Hardware Description Language) code based on an architecture description language LISA 2.0 [40]. The software tools are used to estimate code size and execution time, and the HDL code is used as an input to Synopsys Design Compiler [40] to retrieve the information in terms of hardware costs such as clock cycle time and cell area as shown in Figure 4.8.

In our experiments, we have selected one of the diverse processor models offered by Synopsys Processor Designer as a baseline architecture, which is composed of 4-issue slots, i.e., 4-way VLIW architecture, with two integer ALUs, one floating-point

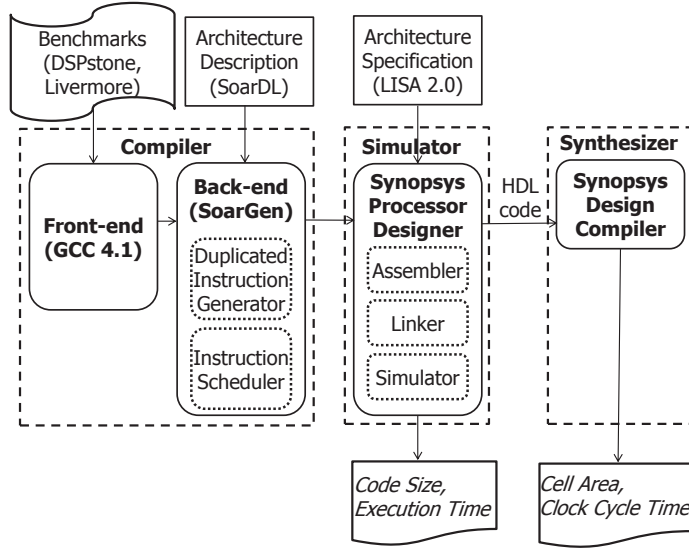


Figure 3.6 Our Compiler-Simulator-Synthesizer Framework

ALU, one load/store unit, and one branch unit. The baseline architecture has a typical RISC-style ISA such as MIPS ISA [9]. For comparison, both the static code duplication architecture and our proposed architecture have been modeled and implemented upon this baseline architecture. After implementing the static code duplication architecture, our proposed architecture has been modeled by modifying the instruction fetch logic to support our compiler-assisted dynamic code duplication scheme as described in Section 3.2.2. Architectures for our compiler-assisted dynamic code duplication scheme and the static code duplication scheme are shown in Figure 3.1.

The compiler for the proposed VLIW architecture is generated by a re-targetable compiler platform, SoarGen [82]. Given the ISA of a target processor architecture written in an architecture description language, SoarDL [82], SoarGen can generate the compiler for the target processor. The proposed vulnerability-aware duplication algorithm explained in Section 3.3.2 has been implemented in the compiler for our proposed VLIW architecture.

Table 3.3 Cell Area Comparisons (unit: μm^2)

Duplication Scheme	Static Code Duplication Scheme	Compiler-assisted Dynamic Code Duplication Scheme
Combinational Area	426635.84	437481.09
Non-combinational Area	136997.53	144327.28
Total Area	563633.37	581808.37

For extensive simulations, we have used two suites of benchmarks, DSPstone [83] and Livermore Loops [84], to evaluate the effectiveness of our approach. DSPstone is kernel benchmarks consisting of code fragments or functions which are commonly used in DSP algorithms. Livermore Loops is a benchmark suite for parallel architectures and consists of a set of loop kernels in numerically intensive applications. To estimate the overhead of our proposed architecture, cell area, power, and clock cycle time are estimated. To show the efficiency of our proposed architecture, the analysis result in terms of code size and execution time will be presented in Section 3.4.2. Also, the effectiveness of the proposed vulnerability-aware duplication algorithm in terms of vulnerability will be presented in Section 3.4.3.

3.4.2 Effectiveness of Compiler-assisted Dynamic Code Duplication

To see the area cost, our first analysis has synthesized HDL codes for both architectures of the static code duplication scheme and our compiler-assisted dynamic code duplication scheme. Table 3.3 shows the logic synthesis results in terms of cell areas from Synopsys Design Compiler with the input of the HDL code generated by Processor Designer in Synopsys [40]. Total cell area includes combinational area and non-combinational area. Compared to that of the static code duplication scheme, the total cell area of our scheme increased by about 3.2%. This overhead results from adding a new pipeline stage due to the split of the fetch stage.

The power consumption overhead caused by adding an extra pipeline stage is negligible at least from our experimental analysis. Since it does not incur much power

overhead in case of the number of pipeline stages 5 to 6 where our architecture has been designed. In our preliminary experiments with McPAT [85], we have estimated power consumptions as the number of pipeline stage increases from pipeline depth 4 to 10 by 1 for all available architectures such as Niagara, Alpha, and X86. We also estimate the power consumption with the regression line from the number of pipeline stages 4 to 10, and it also shows nearly linear regression in range of 3.3% as one pipeline stage is added. In summary, adding one stage to the pipeline is not a big concern in terms of power consumption in this analysis.

We have adopted the energy consumption estimation model from that of the previously proposed static scheme [11]. Our experimental results clearly show that our approach does not incur much overhead in terms of energy consumption as compared to the static scheme. Energy consumption model in the static scheme only considers those of functional units to estimate the impact of duplicating instructions on energy consumption. Our energy consumption estimation is identical to the static scheme except that additional pipeline stage increases the power consumption in our scheme. Since power consumption overhead for our new architecture is significantly small, the energy consumption overhead is also small. Our experiments show minimal energy consumption overhead (less than 3% on average). In summary, our dynamic duplication scheme can be implemented with minimal energy consumption overhead as compared to the static scheme.

Our first set of experiments is to evaluate code size and execution time when duplicating instructions. Figure 3.7 clearly shows the effectiveness of our scheme in terms of code size. Our dynamic code duplication scheme can reduce code size overhead by allocating duplicated instructions at run-time at the cost of an acceptable hardware overhead (about 3.2% area overhead). It is significantly efficient since it does not incur any overhead in terms of memory space for the code. Figure 3.7 shows the code size of our dynamic code duplication scheme as compared to that of a previously proposed

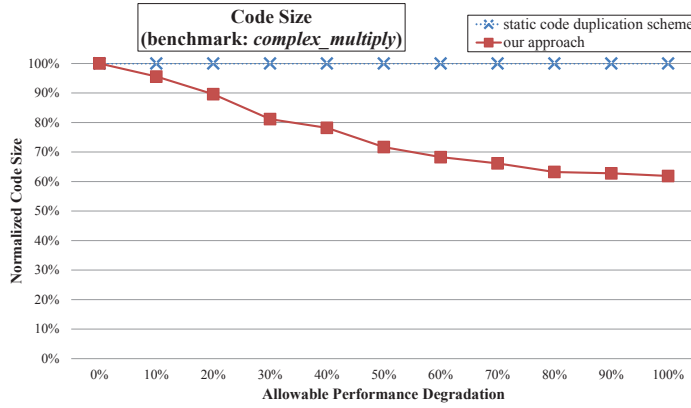


Figure 3.7 The effectiveness in code size reduction

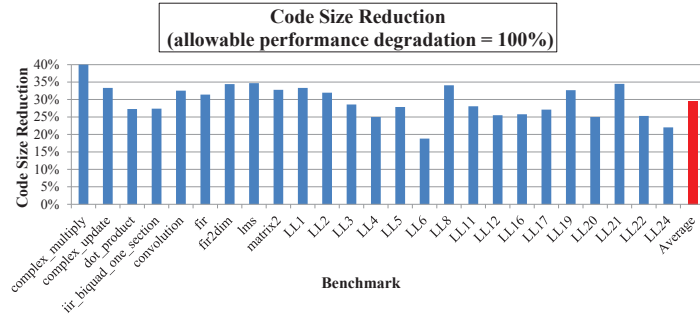


Figure 3.8 The effective code size reduction over benchmarks

static code duplication scheme [11] with *complex_multiply* benchmark of DSPstone suite [83]. Clearly, our approach can achieve the reduction of code size, i.e., it does not incur any duplicated code at compile-time while the static code duplication scheme increases the code size by duplicating more instructions. With the increase of allowable performance degradation, ours can keep reducing the code size and achieve code size reduction by up to about 40% under 100% performance constraint as compared to the static code duplication scheme. This is because our method does not increase code size, while the static code duplication scheme increases code size as more performance degradation is allowed. We can observe similar results with other benchmarks

as shown in Figure 3.8. Our scheme can reduce the code size by about 30% on average over the benchmark suite as compared to the static code duplication scheme in case of full instruction duplication under the 100% performance bound. The main reason of this code size reduction is that our scheme can duplicate instructions at run-time rather than at compile-time.

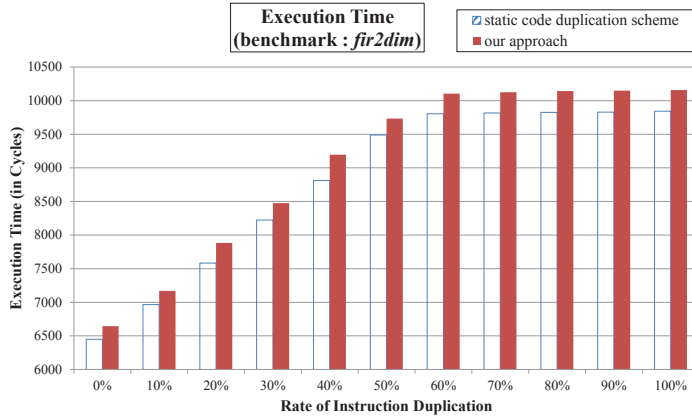


Figure 3.9 The competitiveness in performance

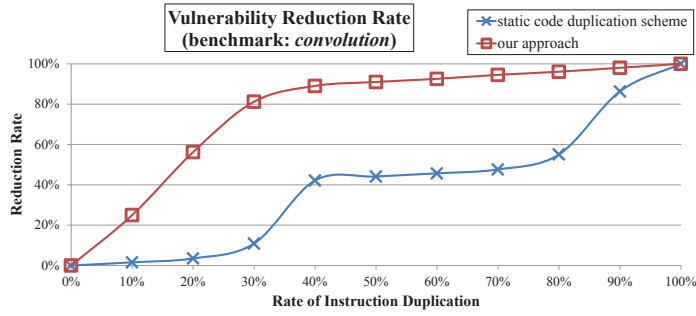
Figure 3.9 represents the result of the performance comparison between our approach and the static scheme in case of *fir2dim* benchmark from DSPstone suite. Our scheme achieves comparable performance with the static code duplication scheme when duplicating the same number of instructions. To estimate performance, the execution time is measured by cycle-accurate simulator from Synopsys Processor Designer [40]. As explained in Section 3.2.1, the duplication range of our scheme is limited to only two cycles to reduce the complexity overhead of hardware implementation while that of the static scheme is the end of a basic block. Since the static scheme can definitely browse the larger interesting space than ours, we also speculated that duplicate instructions of the static scheme would be located to the code with smaller amount of overhead in cycles than our scheme. However, our limited duplication range does not incur significant performance overhead as compared to the static scheme mainly

because most duplicate instructions are allocated within the next cycles of their originals mainly due to data dependency even with the static scheme. We believe that the data dependency prevents the large duplication range of the static scheme from being fully utilized. Indeed, in the static scheme, most duplicate instructions are allocated within the next or a few further cycles of their originals, not the end of the basic block, due to the data dependency. Our experimental results over benchmark suites show that our scheme incurs performance overhead by up to 3% (in case of *fir2dim* benchmark as shown in Figure 3.9) and 1% performance overhead on average, as compared to the static scheme in terms of the execution time. As a result, a small duplication range of our approach incurs negligible performance overhead, which provides a solid foundation for our scheme in terms of performance.

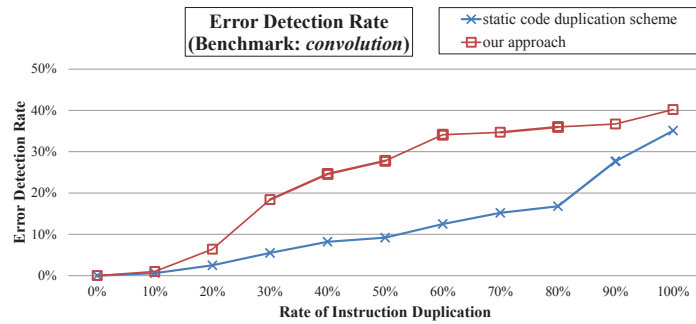
In summary, our scheme can effectively reduce the code size with little performance overhead since our compiler can generate duplication information and our modified fetch stage can utilize this information for instruction duplication at run-time.

3.4.3 Effectiveness of Vulnerability-aware Duplication Algorithm

Our second set of experiments is to evaluate the effectiveness of our approach in terms of vulnerability reduction and error detection. To estimate the amount of vulnerability reduction, our heuristic method quantifies the vulnerability of an instruction using the cell area computed by Synopsys Design Compiler as stated in Section 3.3.2, based on the observation that the larger area occupied by the instruction, the more vulnerable it is [72, 67]. This quantification enables us to reflect *physical vulnerability*. Total amount of vulnerability reduction in a program is the sum of the vulnerability values of all duplicated instructions. The total vulnerability reduction is measured at run-time, not compile-time, to take into account actual loop execution count. This run-time vulnerability measurement makes it possible to reflect *temporal vulnerability* in our study. To validate the effectiveness of our approach, fault injection experiments have



(a) Vulnerability Reduction



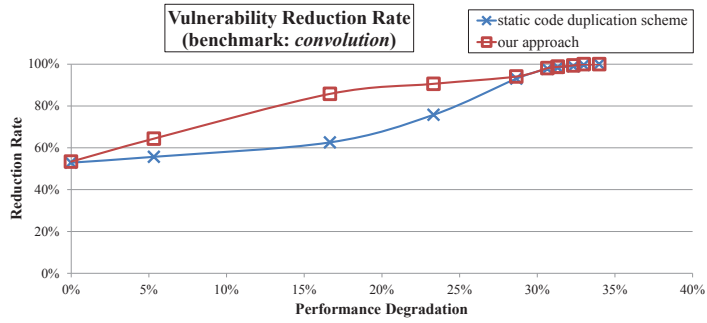
(b) Error Detection

Figure 3.10 Effectiveness of our Dynamic Code Duplication with TVAD in terms of Vulnerability and Error Detection

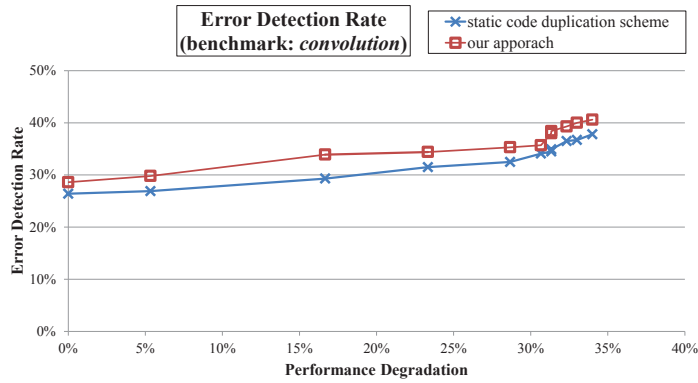
been performed. We have injected soft errors, i.e., random bit errors, into instructions in our simulation framework. A fault rate is one fault per 100 cycles on average with a random time interval between consecutive faults. We have run 1,000 times of each benchmark and calculated the error detection rate which is the number of detected errors over 1,000 experiments. Note that vulnerability reduction experiments show the relative reliability based on a heuristic method and our fault injection experiments show the practical effects of our approach in reliability.

Figure 3.10 clearly shows the effectiveness of considering temporal vulnerability in our approach. For convenience, we call an instruction in a loop, a *loop instruction*, and otherwise, a *normal instruction*. Figure 3.10(a) shows the rate of vulnerability re-

duction according to the change of the number of duplicated instructions for each approach. The rate of vulnerability reduction (R) is calculated as $R = 100 \times (V_{Dup}/V_{All})$ where V_{Dup} is the sum of vulnerabilities of duplicated instructions and V_{All} is the sum of vulnerabilities of all duplicable instructions. When a loop instruction is selected for duplication, its vulnerability is accumulated repeatedly as much as dynamic loop execution count in computing V_{Dup} . For a fair comparison, the rate of vulnerability reduction is estimated under the constraint that both approaches have identical numbers of duplicated instructions. Our approach achieves more vulnerability reduction than the static code duplication scheme since a loop instruction has higher priority for duplication than a normal one and therefore more loop instructions can be duplicated in our approach than the static code duplication scheme if the same number of instructions are duplicated. On the other hand, it is likely that more normal instructions are duplicated in the static code duplication scheme since it duplicates instructions in the order of their locations in the code. Thus, the static code duplication scheme achieves less vulnerability reduction than our approach. In *convolution* of DSPstone benchmark, our approach can achieve more vulnerability reduction by up to 70% and by on average 43% than the static code duplication scheme. Figure 3.10(b) shows the error detection rate according to the change of the number of duplicated instructions for each approach. The error detection rate is calculated as the ratio in percentage of the number of detected errors to the total number of the injected errors in our simulation framework. Figure 3.10(b) shows that the error detection rate follows the trend close to the vulnerability reduction rate with a benchmark, *convolution*. Our approach can achieve more error detection by up to 22% and by on average 13% than the static code duplication scheme. Note that our approach seems less effective in terms of error detection than in terms of vulnerability reduction and it is mainly because an error can be easily injected on instructions without their duplications and an injected error does not always result in different outputs at the comparison.



(a) Vulnerability Reduction

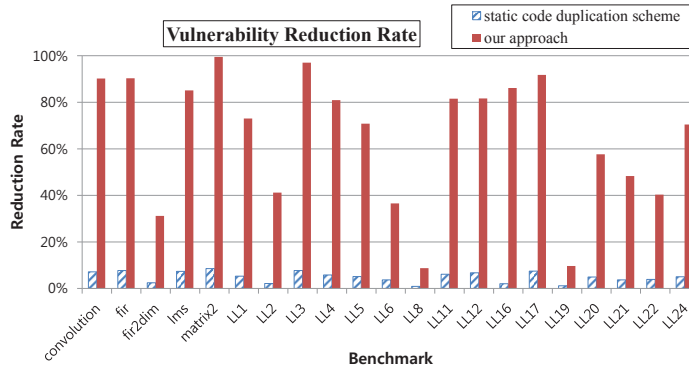


(b) Error Detection

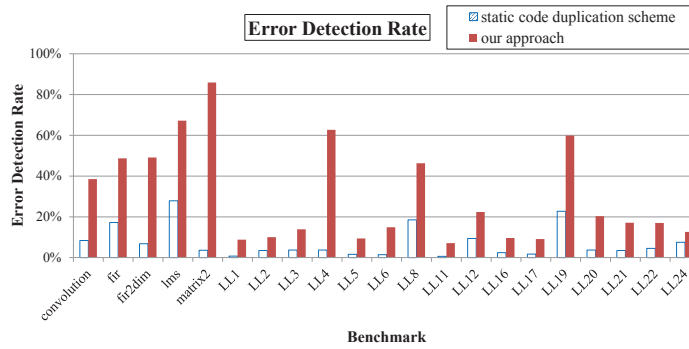
Figure 3.11 Effectiveness of Our Dynamic Code Duplication Scheme with PVAD in terms of Vulnerability and Error Detection

Figure 3.11 clearly shows the effectiveness of considering physical vulnerability in our approach. By our heuristic method, each instruction has its own physical vulnerability based on the cell area occupied by itself as explained before. Figure 3.11(a) shows the rate of vulnerability reduction according to the change of performance degradation. Note that the performance degradation is a knob to adjust the reliability by duplicating instructions under that performance bound. In case of the maximal performance degradation, both approaches achieve very close vulnerability reduction since most instructions are duplicated. However, when performance degradation is limited, i.e., a small number of instructions could be duplicated, our approach can achieve

more vulnerability reduction than the static code duplication scheme. This is because the instructions with higher physical vulnerability can be preferentially duplicated in our approach while the static code duplication scheme cannot. In *convolution* of DSPstone benchmark, our approach can achieve more vulnerability reduction by up to 23% and by on average 5% than the static code duplication scheme. Figure 3.11(b) shows the error detection rate according to the change of performance degradation. In *convolution* of DSPstone benchmark, our approach can detect more errors by up to 5% and by on average 3% than the static code duplication scheme.



(a) Vulnerability Reduction



(b) Error Detection

Figure 3.12 Effectiveness of Our Dynamic Code Duplication with TPVAD in terms of Vulnerability and Error Detection

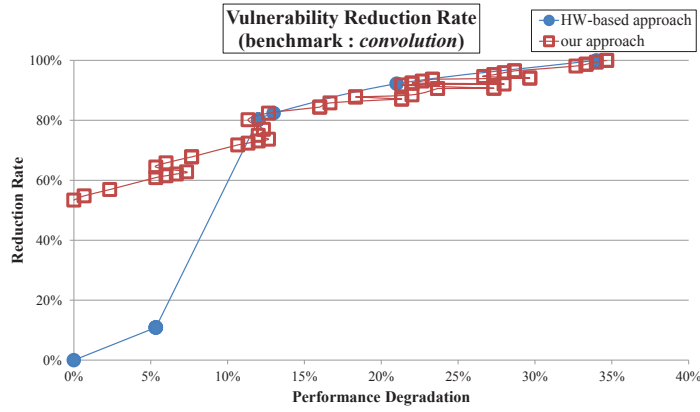
Figure 3.12 clearly shows the effectiveness of our approach in terms of vulnera-

bility reduction and error detection when jointly considering both temporal and physical vulnerability. Figure 3.12(a) shows the rate of vulnerability reduction for both approaches when the difference between the rates of vulnerability reduction of both approaches is maximum for each benchmark. In common with the case of considering only the temporal vulnerability, the rate of vulnerability reduction is estimated under the constraint that both approaches duplicate the same number of instructions for each benchmark. Obviously, the effectiveness of considering two types of vulnerabilities simultaneously is better than that of considering solely one of those vulnerabilities. Over benchmarks, our approach can achieve more vulnerability reduction by up to 91% and by on average 60% than the static code duplication scheme. Figure 3.12(b) shows the error detection rate for both approaches when the difference between the error detection rates of both approaches is maximum over benchmarks. Our approach can detect more errors by up to 82% and by on average 23% than the static code duplication scheme.

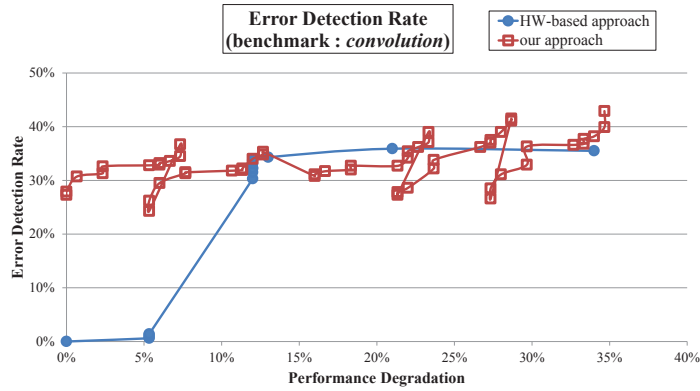
Note that our vulnerability reduction measurement is an approximate estimation. To achieve more accurate estimation of vulnerability reduction, AVF (Architectural Vulnerability Factor) can be used as a metric. However, since our proposed approach is orthogonal to methodologies for measuring the vulnerability reduction, our experimental results are enough to show the relative efficiency of our approach even if there might exist some degree of incorrectness. This is why fault injection experiments have been performed and error detection rate has been calculated.

In summary, our approach can reduce the vulnerability and increase error detection rate, i.e., increase reliability effectively since ours can consider the importance of instructions when selecting instructions to be duplicated under the performance constraint.

Our third set of experiments is to show the efficacy of our compiler-based dynamic code duplication scheme in terms of vulnerability as compared to the hardware-based



(a) Vulnerability Reduction



(b) Error Detection

Figure 3.13 The effectiveness of our compiler-assisted dynamic code duplication as compared to hardware-based approach

dynamic code duplication scheme. Figure 3.13 clearly shows the effectiveness of considering the instruction vulnerability at software level. Unlike the software-based approach where the instruction vulnerability is considered at the software level such as our proposal, X. Vera et al. [72] present a hardware-based approach that considers the instruction vulnerability at hardware level as explained in Section 3.1. Figure 3.13(a) shows that there is an explicit trade-off between the rate of vulnerability reduction and the performance in the hardware-based approach. However, our approach can achieve

more vulnerability reduction when a small performance overhead is allowed as compared to the hardware-based approach. Similarly, Figure 3.13(b) shows that our approach can detect more errors than the hardware-based approach when a small performance is allowed. This is because our approach can control the threshold value according to the characteristics of instructions such as whether they exist in a loop or not, and thus achieve as more vulnerability reduction as possible at compile-time.

In summary, our approach can present a more flexible solution for improving the system reliability compared to the hardware-based approach.

Chapter 4

Selective Validation Techniques for Robust CGRAs against Soft Errors

In order to address soft error resilient CGRAs with the least performance overhead, we propose software implemented redundancy techniques on CGRAs with selective validation mechanisms. First, we identify the expensiveness of validation mechanisms for TMR and DMR on CGRAs, respectively. Indeed, the voting overhead takes up approximately 64.8% of the total overhead in TMR and it is true since CGRAs are good at data intensive computation rather than control intensive computation such as voting operations [1]. Second, we present selective validation schemes for software based TMR and DMR on CGRAs rather than the complete validation. The main idea behind our proposals is to selectively apply voting mechanisms just before synchronous points where applications can be affected by corrupt data induced by soft errors and fail to deliver the correct results. Also, we present the comparable fault coverage of our approach as compared to the previously proposed software-based TMR technique with the full voting on CGRAs. In addition, we propose an optimization technique to reduce the synchronous points so that we can further reduce the performance and en-

ergy consumption overhead due to the complex voting by decreasing the number of voting mechanisms.

The contribution and results of this work include :

- Our software based TMR technique with the selective voting can improve the runtime by 38.3% and the energy consumption by 18.1% on average over benchmarks as compared to a previously proposed TMR technique with the full voting.
- Our software based DMR with the selective comparison can improve the runtime by 14.3% and the energy consumption by 3.6% on average over benchmarks as compared to a previously proposed DMR technique with the full comparison.
- Our optimization techniques can further improve the runtime by 41.0% and the energy consumption by 26.2% as compared to a previously proposed TMR technique with the full voting mechanism and by 17.8% and 14.0%, respectively, as compared to a previously proposed DMR technique with the full comparison mechanism by minimizing the occurrence of the validations with the loop unrolling scheme.
- Our software based protection techniques with the selective validation mechanism show the fault coverage as comparable as recent proposals with the full voting mechanism by quantitative analysis.

4.1 Related Works

Due to the high performance and programmability, CGRA is becoming more popular as the alternative of existing accelerators such as GPGPU (General-Purpose computing on Graphics Processing Units), FPGA, etc. CGRA architectures can accelerate various kernels of intensive applications such as multimedia applications and software defined radio (SDR) by exploiting reconfigurability. Since performance and programmability are two important and challenging issues, most of researches on CGRAs have been focused on the compilation techniques and scheduling algorithms.

However, soft errors are becoming a critical design concern as technology scheduling and CGRA is being employed in critical applications such as aircrafts, space missions, and financial systems [27]. Thus, the reliability on CGRAs against soft errors is emerging as an important research topic but the literature is relatively small. Most of these studies proposed redundancy techniques such as TMR and DMR by exploiting identical blocks or processing elements on CGRAs for the replications. First, most of researches improve the reliability by hardware modification like TMR and DMR. However, hardware-based techniques incur the huge area overhead caused by additional hardware logic to implement replications and comparison, voting mechanisms. Second, several researches exploit the software-implemented fault tolerant techniques in order to overcome hardware overhead. However, it still induces the significant overhead in terms of performance due to replication and validation.

Alnajjar et al. [28] proposed dynamic operation modes in CGRA architecture to provide the various levels of reliability under the performance constraint. However, their technique incurs 26.6% area overhead mainly due to additional hardware redundancy and causes performance degradation because cluster-based architectures cannot fully use hardware resources. To reduce this hardware overhead, Jafri et al. [27] presented an alternative hardware-based redundancy technique, residue mode less costly than DMR to detect soft errors. They implemented self-checking residue mode for multiplication and addition operations on DART architecture [86], but it cannot be applied to logical operations. Recently, Eisenhardt et al. [87] proposed the remapping engine process designed and suitable for permanent faults, not soft errors.

On the other hand, researchers have investigated different approaches from the previously proposed techniques that redesign and modify architectures of processing elements on CGRAs to reduce the hardware cost. Kim et al. [88] observed that not all processing elements are exploited at the execution time mainly because some of processing elements are used for the routing of operands between producing and consuming operations. Based on this observation, Schweizer et al. [30] proposed techniques exploiting unused FUs for replications to increase the reliability with the minimal hardware overhead. They proposed FEHM (Flexible Error Handling Module) that supports TMR and DMR schemes on specific target architectures. However, data intensive application cannot map all the operations to processing elements due to insufficient unused FUs. To resolve this limitation, they introduced multiple contexts to be mapped on CGRA by using the concept of temporal redundancy [29]. However, the increased number of the contexts incurs 12% performance degradation and there still remains unresolved hardware overhead. In short, previously proposed hardware based techniques incur additional area cost since they need to modify existing CGRA architectures to implement redundancy techniques such as TMR and DMR.

In order to overcome this area overhead, recent researchers have investigated software based techniques to implement redundancy techniques without hardware mod-

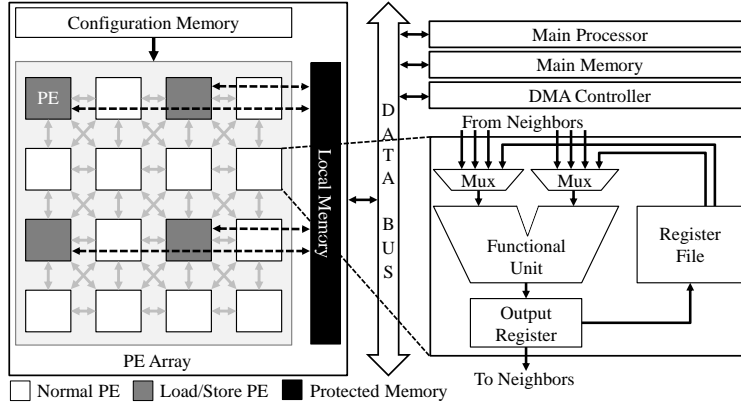


Figure 4.1 Our 4×4 CGRA architecture

ifications [33, 1]. Singh et al. [33] presented fault tolerance techniques for an existing Raw architecture [89] by exploiting the selective redundancy and checkpoint schemes. However, their scheme is inapplicable to any CGRA architecture since it is designated only for RAW architecture and also causes performance degradation. As a general software-implemented technique applicable to any CGRA architecture, Lee et al. [1] proposed software based TMR and DMR techniques by mapping software-implemented replicas of operations and validation mechanisms onto processing elements in CGRA architectures. However, they still incur high performance overhead mainly due to additionally mapped processing elements for the complex voting and comparison mechanisms. These software techniques offer limited amount of area cost than hardware techniques but result in significant performance degradation.

We propose novel selective validation schemes to improve performance without any hardware modification. Our proposals can remove the area cost by exploiting software based techniques and fulfill the performance improvement by selectively applying the validation mechanisms only on synchronization points before *store* operations. This approach makes sense since modified outputs from processing elements (which will be written back to the memory) can affect the application kernel and its final output at the end unless they are fixed before the memory update [64].

4.2 Motivation

CGRA is essentially an array of processing elements or PEs connected through a mesh-like or interconnection as illustrated in Figure 4.1. A PE generally consists of a functional unit (e.g., ALU, shifter, multiplier, etc.) and a small register file for storing temporary variables and constant values. The PE array consists of heterogeneous PEs

```

for ( i = 0; i < iteration; i ++ ) {
  /* X and Y are constants */
  a[i] = ( b[i] - X ) / Y;
}

```

(a) Example of a kernel

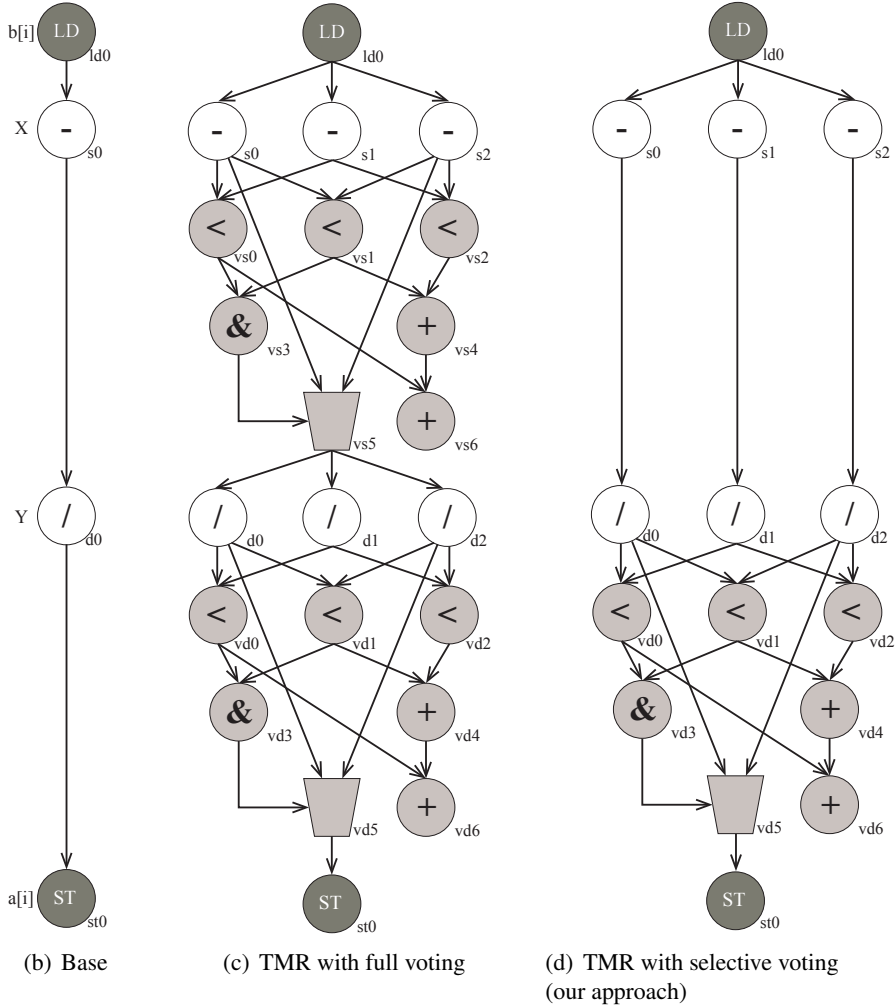


Figure 4.2 A simple kernel and its generated DFGs of base (no redundancy), software implemented TMR with the full voting [1], and TMR with the selective voting (our approach)

and basic operations such as arithmetic and logical operations are performed by every PE whereas the costly operations such as multiply and memory access operations are performed only by some PEs. Like Field-Programmable Gate Arrays (FPGAs), the functionality of PEs and the data flow among PEs are controlled by configuration. However, as the configuration size for CGRA is small since CGRAs are controlled in a word-level operation, CGRAs can be reconfigured very fast, even in every cycle [90], unlike FPGA configured in a bit-level operation.

Figure 4.2 shows an example of software-implemented TMR for a kernel part. Figure 4.2(a) presents C-like pseudo code for the kernel part ($a[i] = (b[i] - X)/Y$). To implement this kernel, the original DFG (Data-Flow Graph) is composed of *load*, *subtract*, *divide*, and *store* operations as shown in Figure 4.2(b). Figure 4.2(c) draws its TMR implementation as a DFG form. In this example, normal operations (except memory operations) such as *subtract* and *divide* must be triplicated and their results are validated by the voting mechanism. For instance, a *subtract* operation (the original node, s0) is triplicated (two additional nodes s1 and s2 for the triplication) and the voting requires three *compare* (vs0, vs1, and vs2), one *logical and* (vs3), two *add* (vs4 and vs6), and one *select* (vs5) operations (7 additional nodes for the voting). A TMR implementation requires 9 additional nodes per normal operation, which is translated into the huge impact on the performance. For this simple kernel, TMR implementation has increased the number of nodes from 4 to 22 and the number of edges from 3 to 35 (compare Figure 4.2(b) and Figure 4.2(c)). These increased numbers of nodes and edges increase the complexity of the operations to be mapped onto PEs causing more challenges to the compilation scheduling. Eventually, they degrade the performance due to highly required PEs and tightly induced data dependency among them.

In order to observe the performance overhead of the voting mechanism, we have run a simple experiment. First off, we have evaluated the performance in terms of runtime for base kernels (i.e., without any redundancy) of benchmarks. Secondly, the runtime (explained in Section 4.4.1) has been estimated for software based TMR implementation on CGRA and its performance overhead has been calculated in percentage by dividing the difference between runtime of the base and that of the TMR by that of the base ($O_{TMR} = (R_{TMR} - R_{Base})/R_{Base}$ where O_{TMR} is its performance overhead, and R_{Base} and R_{TMR} are runtime for the base and TMR with the full voting, respectively). Then, we have implemented the DFGs of triplicated operations of benchmark kernels without the voting, and evaluated the runtime and its performance overhead ($O_{TMR_no_vote} = (R_{TMR_no_vote} - R_{Base})/R_{Base}$ where $O_{TMR_no_vote}$ is its performance overhead and $R_{TMR_no_vote}$ is runtime for TMR without the voting.). Last, we estimate the voting overhead as the difference between O_{TMR} and $O_{TMR_no_vote}$, and Figure 4.3 draws the portions of the voting overhead ($O_{TMR} - O_{TMR_no_vote}$) and the triplication overhead ($O_{TMR_no_vote}$). Figure 4.3 shows that the performance overhead caused by the voting mechanism in software-implemented TMR takes up

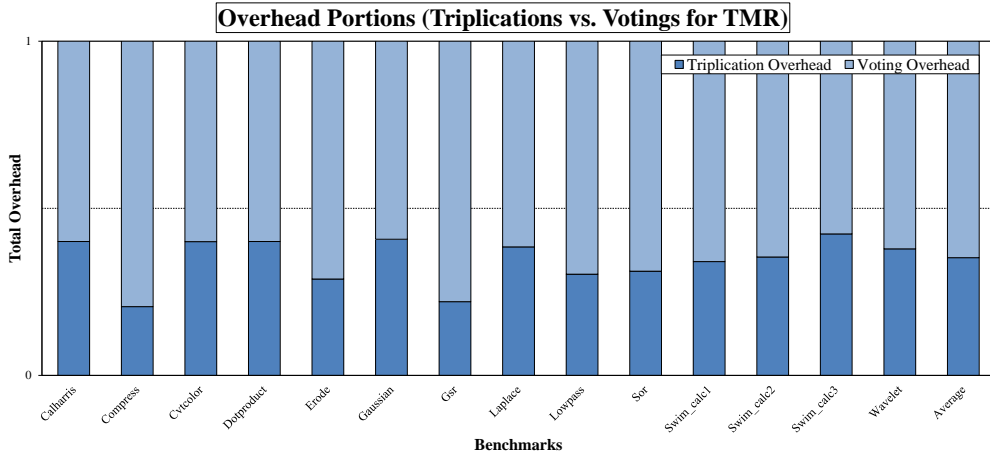


Figure 4.3 Runtime overhead of voting overhead takes up about 64.8% in software implemented TMR techniques on CGRAs.

about 64.8% on average over the benchmarks. This high performance degradation results from the high data dependency and complexity of the voting mechanism. In this dissertation, I investigate selective validation techniques to reduce this expensive performance cost in redundancy protections on CGRAs.

4.3 Our Approach

4.3.1 Selective Validation Mechanism

In order to improve the reliability with minimal performance overhead, we present the selective validation techniques for TMR and DMR on CGRAs. Our goal is to protect the datapath of CGRAs such as FUs from soft errors. Traditional hardware-based protection methodologies for memory subsystem are inexpensive as compared to maintaining double- or triple-redundant execution cores [64]. Therefore, we suppose that memory of CGRA architectures is protected against soft errors by traditional fault tolerant techniques such as parity check, ECC (Error Correction Code), and scrubbing. Thus, we do not replicate the memory operations such as *load* and *store* and do not validate the output of memory operations, which can reduce the performance overhead.

Our main goal is to reduce the number of validations for improving performance without losing the reliability as compared to the existing redundancy techniques. Note that the main benefit of CGRAs is to map the kernel part of applications to accelerate the performance that is data intensive kernel as in operations in the loop. Thus, the control part of applications is not suitable for being mapped onto the PEs in CGRAs

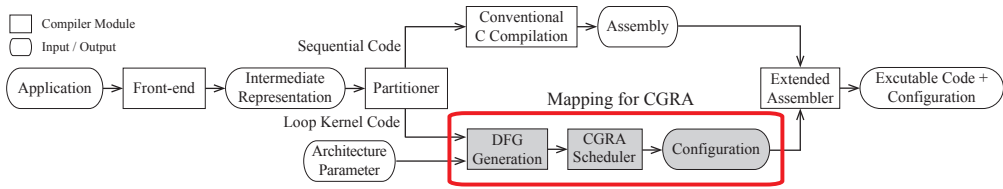
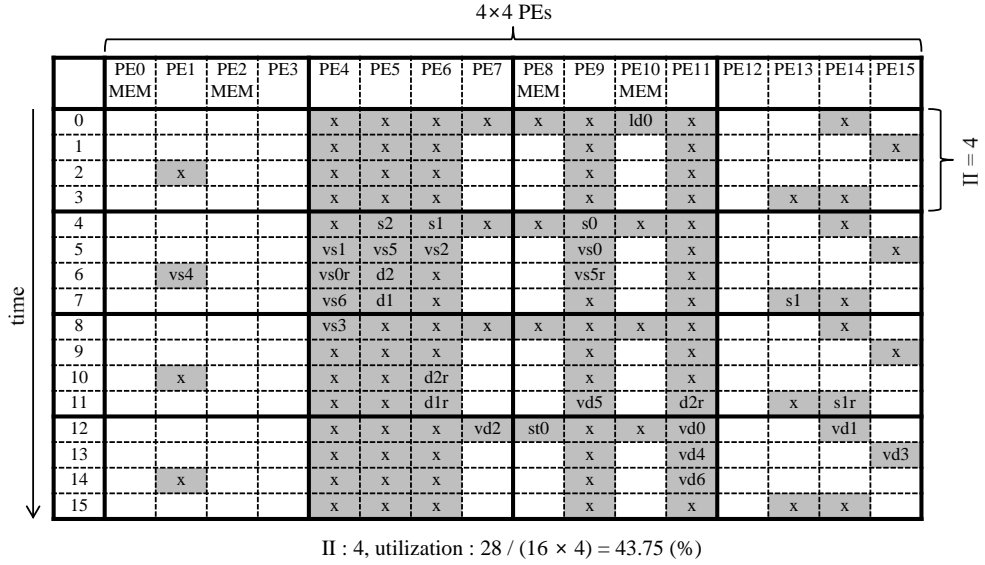


Figure 4.4 Compilation flow for a system with CGRA

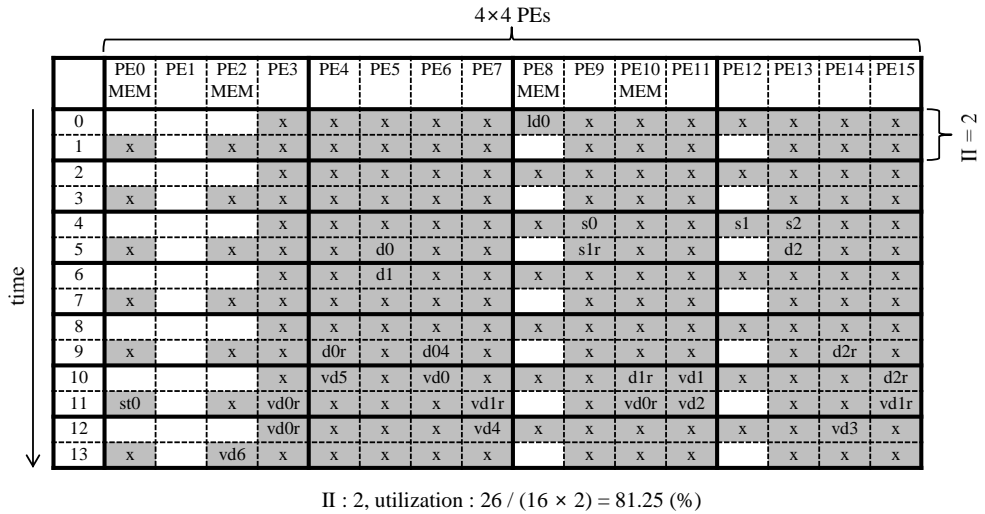
since it incurs unnecessary performance overhead. Indeed, the validation mechanisms such as the voting mechanism for TMR and the comparison for DMR are sort of control intensive operations which are inappropriate for operations mapped onto PEs in terms of the performance. The main idea behind our approach is to perform validation operations just before synchronization points where the program can be affected and result in incorrect output or even catastrophic consequences if the data is incorrect after synchronization points [64, 91]. For example, *store* operations have been committed to the memory with erroneous states and these erroneous results can eventually cause incorrect outputs of an application. Thus, the program will be executed correctly if corrupted data is not stored in the main memory. Indeed, the concept of this selective validation approach has been introduced through the technique named SWIFT (Software-Implemented Fault-Tolerant) [64, 91]. In this technique, all instructions other than memory instructions are replicated and the validation checks are introduced only at certain synchronization points to ensure that the data produced by the original and replicated operations are identical or correctable. Note that their technique can achieve the reliability by 97% of that of TMR with the full validation [91].

4.3.2 Compilation Flow and Performance Analysis

Figure 4.4 shows the overall compilation flow for a system including CGRA as an accelerator or coprocessor. First, an application is partitioned to extract kernels to be mapped onto CGRAs. Then, the extracted kernels are compiled for CGRA while the rest of the code, i.e., sequential code, goes through the conventional compilation process. CGRA compilation starts from constructing the DFG of a loop. After that, modulo scheduler takes the DFG as an input to generate a valid mapping result for executing the loop on CGRA. Modulo scheduling [92] is a software pipelining technique that exploits the parallelism by overlapping consecutive iterations of the loop. The goal is to find a valid schedule with a minimized initiation interval (II), which is the difference between the start times of successive iterations. Minimizing the II leads to the throughput improvement since one loop iteration takes II cycles ignoring the effects of prologue and epilogue of the loop. Modulo scheduler first initializes the II by taking the maximum out of the *resource-constrained lower bound* (ResMII) and



(a) TMR implementation with full voting from Figure 4.2(c)



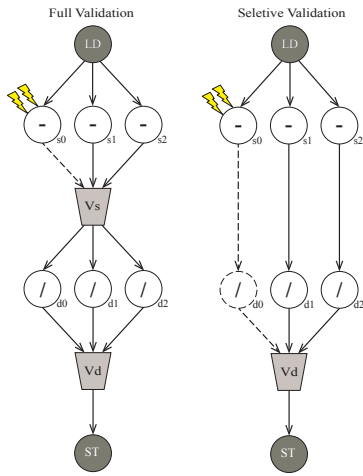
(b) TMR implementation with selective voting from Figure 4.2(d)

Figure 4.5 Mapping operations for software implemented TMR onto CGRAs

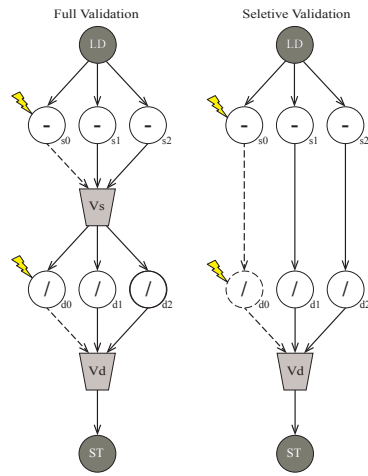
the *recurrence-constrained lower bound* (RecMII). It then attempts to generate a valid schedule within the minimal II. If no valid schedule can be found for the given II, the scheduler increments II by one and attempts again until a valid schedule is achieved.

Figure 4.2 shows the generated DFGs through this compilation flow for the original code (Figure 4.2(b)), for the TMR code with the full validation (Figure 4.2(c)), and for the TMR code with the selective validation (Figure 4.2(d)). Our TMR with the selective validation introduces just one validation computation as shown in Figure 4.2(d) while TMR with the full validation introduces two validation computations as shown in Figure 4.2(c). Thus, we can reduce one set of operations for the validation after the triplicated operation (*subtract* operation in this example), and this reduction can improve the performance.

Figure 4.5 illustrates the effectiveness of our selective voting technique in terms of II and the utilization with a mapping example. Each DFG is scheduled onto a 4×4 CGRA according to our compilation flow in Figure 4.4. In the scheduled results, the ID at each cell in Figure 4.5 indicates the mapping of an operation from the DFG as shown in Figure 4.2. For instance, 's0' is scheduled onto PE 9 at the cycle 4 in Figure 4.5(a). The IDs followed by 'r' (e.g., 'vs0r') indicate routing operations for the corresponding computation operations. For example, 'vs0r' is scheduled onto PE 4 and at the cycle 6 in Figure 4.5(a) for the routing of 'vs0'. Slots marked with 'X' represent ones occupied due to the modulo constraint. Assume that the latency of a *load* operation is three cycles and other operations one cycle in our scheduling framework. Figure 4.5(a) shows the scheduling result for TMR with the full voting consisting of 22 nodes and 35 edges from the DFG in Figure 4.2(c), and its performance output with II=4. However, our selective voting technique can construct the DFG with less nodes and edges (15 nodes and 21 edges from Figure 4.2(d)) and thus the II from the scheduled result is equal to 2 (as shown in Figure 4.5(b)), which can be interpreted about 2 times improvement in performance since the performance is roughly proportional to the II. Interestingly, we can utilize the PEs of CGRA approximately 2 times more efficiently with the selective voting than the full voting. Note that the better utilization can avoid unnecessary waste of CGRA resources and lead to the performance efficiency. Therefore, our selective voting technique can improve II and archive high utilization ratio due to the reduced number of nodes and edges. The number of nodes and edges in the DFG affects several aspects as follows. First, the number of nodes implies the least required number of PEs in CGRA architectures. Second, the increased number of edges in general raises the data dependency among connected PEs. Our selective validation techniques can improve the performance with the reduced II by decreasing the number of nodes and edges in the DFG and by exploiting the unused FUs efficiently by reducing the data dependency.

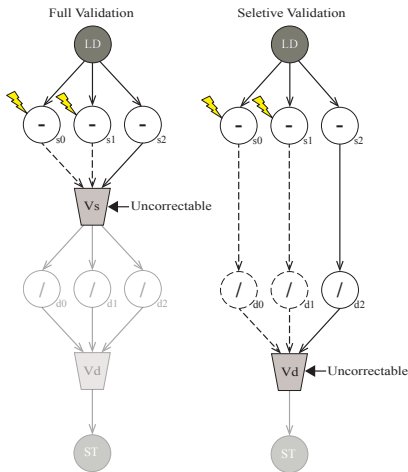


(a) Case 1: two soft errors in $s0$

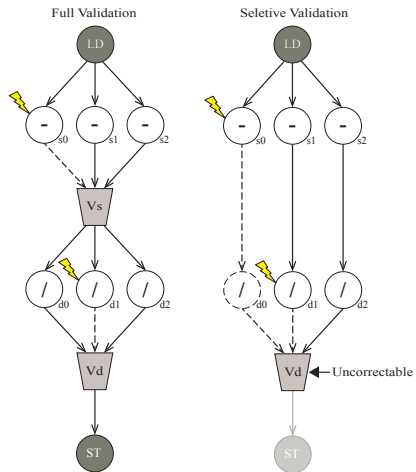


(b) Case 2: one soft error in $s0$ and one soft error in $d0$

- -> Erroneous Data -> Correct Data ⚡ Error Occurrence () Node with Error Propagated



(c) Case 3: one soft error in $s0$ and one soft error in $s1$



(d) Case 4: one soft error in $s0$ and one soft error in $d1$

Figure 4.6 Fault coverage analysis of software implemented TMRs in case of double soft errors (faded nodes and edges indicate no more execution in case of uncorrectable error detection at validations in Figure 4.6(c) and Figure 4.6(d))

4.3.3 Fault Coverage Analysis

Our redundancy techniques with the selective validations on CGRAs can achieve the comparable reliability in terms of the fault coverage as compared to the redundancy techniques with the full validations. Fault coverage can be defined as the ratio of the detected number of faults to the total number of faults. Suppose that the fault coverage of considering both single bit soft errors and multiple bit soft errors is $FC = \alpha \times FC_{SBSE} + \beta \times FC_{MBSE}$ where FC_{SBSE} is the fault coverage for single bit soft errors, FC_{MBSE} is the fault coverage for multiple bit soft errors, and α and β are weight constants for FC_{SBSE} and FC_{MBSE} , respectively. If we consider a single bit error for the whole operation of the kernel, FC_{SBSE} for TMR with the selective validation is equal to that for TMR with the full validation since a soft error induced incorrect value will be eventually corrected at the synchronous point by the validation, which does not cause data corruption or system failure. Thus, when α is set to 1 and β is set to 0, FC for our technique is the same as that for a conventional TMR technique.

On the other hand, if we consider double bit soft errors as multiple ones, which has extremely lower error rate than single bit soft error (100 times less [93]), 4 cases should be taken into account for the fault coverage analysis as described in Figure 4.6. The first case is that double bit errors occur at the same operation in the datapath at the same cycle as shown in Figure 4.6(a). These errors should be fixed by both techniques, i.e., our selective validation and the full validation since no erroneous datapaths in nodes s1 and d1, and s2 and d2 will mask the error propagated to d0 from s0 at Vd in our selective validation as shown in Figure 4.6(a). Thus, the first case results in the same fault coverage. The second case is that double bit errors occur at different operations in the same datapath at different cycles (Figure 4.6(b)) and then these errors also can be fixed by both techniques since operations at the other datapaths will be executed correctly without errors. Thus, the second case also results in the same fault coverage. The third case is that double bit errors occur at the same operations in two different datapaths at the same cycle (Figure 4.6(c)) and then these errors cannot be validated by both techniques since the 2-out-of-3 voting may not work to mask these errors. Thus, the third case results in the same missed fault coverage. The last case is that double bit errors occur at different operations in two different datapaths at different cycles (Figure 4.6(d)) and then these errors will be corrected by the full validation (since each single bit error can be fixed just after each operation has been committed) but these errors may not be masked by the selective validation. Thus, the last case results in the loss of the fault coverage for the selective validation. Thus, when α is set to 0 and β is set to 1, FC for our technique is worse than FC for TMR with the full validation on CGRAs. Assume that double bit soft error rate is considered 100 times less than single bit soft error rate. If we suppose that all multiple bit soft errors are double bit soft errors and the last case (worse fault coverage case for our technique) takes up the

whole possibility out of four cases, the FC for the selective validation is less than 1% than that for the full validation in TMR, which is the upper bound of the worse fault coverage for our case even in considering various weight constants between 0 and 1 for α and β . In conclusion, our technique can achieve the same fault coverage for single bit soft errors and the comparable fault coverage for multiple bit soft errors (at most 1% worse with the current ratio of single bit soft errors to double bit soft errors) as compared to the previously proposed TMR techniques with the full validation. Note that our fault coverage analysis excludes the cases where soft errors occur on the PEs for the validation mechanisms together. However, an error at the validation cannot guarantee the reliability for both the full and selective validation techniques.

4.3.4 Our Optimization - Minimizing *Store* Operation

To further improve the performance, our optimization technique merges multiple *store* operations into one *store* operation by applying the loop unrolling and modifying the DFG. As illustrated in Figure 4.7, the original loop unrolling can duplicate the DFG to improve the performance. Assume that the data in the same array are stored in adjacent addresses in the memory. Our idea is that the data in adjacent locations will be stored at one access after merging two *store* operations into one by applying *shift* and *add* operations. For example, $a[0]$ is set to 0x12 and $a[1]$ is set to 0x34 in our example as shown in Figure 4.7(a). If the unit size of an array in this example is 1 byte while the variables are of two bytes, $a[0]$ will be shifted by 8. And then $a[1]$ (0x0034) will be added to this shifted value of $a[0]$ (0x1200). Finally, the sum of $a[0]$ and $a[1]$ (i.e., 0x1234) will be stored by just one *store* operation as shown in a form of the DFG in Figure 4.7(c).

After applying our optimization technique, the number of *store* operations can be halved. Therefore, the number of validations also can be reduced in a half so it can improve performance. However, there are two limitations in our optimization technique. First, our optimization requires additional PEs for mapping operations such as *shift* and *add* operations. However, the number of PEs required by the voting mechanism that is 7 greater than that of these additional PEs. Second, our optimization introduces the dependency between unrolled loops. In the original loop unrolling, each unrolled loop can be executed in parallel. In contrast, our approach requires the sum total between the results of these unrolled loops. Therefore, our optimization techniques must be considered with unrolling factors that are number of copied loop kernel. However, determining the unrolling factor with considering CGRA architectures and property of kernels is beyond our scope. Since software-implemented voting requires much more additional nodes than comparison, our optimization technique has the strength in triplication case, rather than duplication.

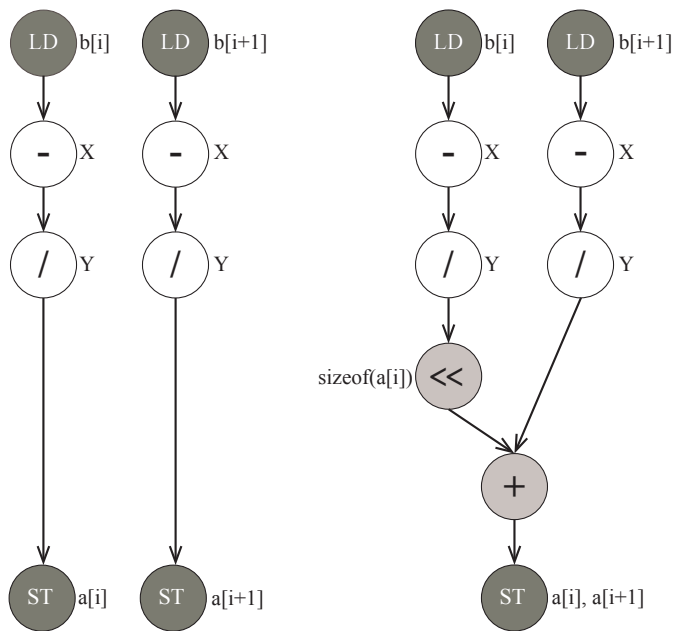
Note that our optimization techniques cannot be applicable for the kernel that has

```

for ( i = 0; i < iteration; i += 2 ) {
    /* X and Y are constants */
    a[i] = ( b[i] - X ) / Y;
    a[i+1] = ( b[i+1] - X ) / Y;
}

```

(a) Example of a kernel



(b) Conventional loop unrolling (c) Modified loop unrolling (our approach)

Figure 4.7 Generated DFGs of our optimization technique with the loop unrolling

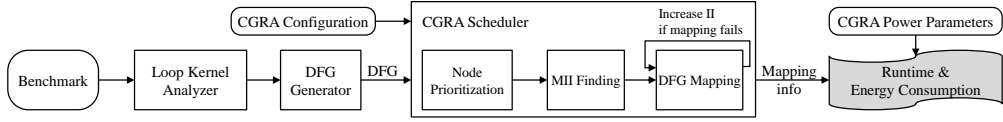


Figure 4.8 Our framework for simulations

recurrent loops. The output of the previous iteration is required as the input of current iteration, so it cannot be stored at the same time. If our CGRA architecture exploits the reuse edge proposed in [94], the output of previous data can be used before *store* operation. However, it must be required the performance overhead tradeoff between voting and exploiting reuse edge techniques. The optimization techniques for recurrent loops are definitely one of our future works.

4.4 Experiments

4.4.1 Setup

To evaluate the effectiveness of our selective protection and optimization techniques, we have implemented a simulation framework as shown in Figure 4.8. For the target architecture, we consider a CGRA that is close to the one illustrated in Figure 4.1. It contains a 4×4 PE array consisting of 4 multiplier PEs, 8 normal operations PEs, and 4 *load-store* PEs. Our CGRA has no shared register file, but each PE has its own register file whose entry size is 8. The local registers are used for scalar variables or routing temporary data. Each PE is connected to its four neighbor PEs, four diagonal ones and 2-hop straight ones. These CGRA configuration is the input to our framework as shown in Figure 4.8.

We have taken important loops as our benchmark suite from multimedia benchmarks, OpenCV benchmarks [95] and SPEC 2000 benchmarks [96]. DFG generator creates a DFG for each benchmark kernel and this DFG information is an input to our compiler and scheduler with an initial *II*. Mapping and routing information of benchmarks onto CGRAs are generated using a version of modulo scheduling [94]. Due to the randomness in the cost-based scheduling algorithm (as there is more than one minimum cost candidate), we compile and simulate each benchmark kernel ten times and the result having minimum *II* among 10 trials is taken as the representative performance for that benchmark. Our experimental framework also returns the runtime in cycles with the minimum *II*.

The runtime is estimated as the sum of the prologue runtime, the kernel runtime, and the epilogue runtime. The prologue runtime R_P and the epilogue runtime R_E are the execution times before and after the kernel execution, respectively, and they are equal to $(s - 1) \times II$ where *II* is the minimum *II* and *s* is the number of stages from

Table 4.1 CGRA Power Parameters

Module	Variable	Power Dissipation (mW)
Active PE (ALU)	P_{ALU}	2.543
Active PE (Multiplication)	P_{MUL}	3.200
Active PE (Division)	P_{DIV}	3.465
Active PE (Routing only)	P_{ROUT}	0.847
Idle PE	P_{IDLE}	0.254
The rest part of PE array	P_{REST}	25.988
Memory bank access	P_{MEM}	270.030
Configuration cache access	P_{CONF}	34.837

our simulations. The kernel runtime R_K is calculated as $(i - s + 1) \times II$ where i is the number of iterations for the benchmark loop. The number of iterations for the *store* reduction i' is calculated as $i/2$ because two operations are merged by loop unrolling as shown in Figure 4.7. The total runtime R is represented as $R_P + R_K + R_E$.

We estimate the energy consumption by using CGRA power parameters as summarized in Table 4.1 [97]. The total energy consumption is estimated as the sum of energy consumptions for the prologue, the kernel, and the epilogue. The summed energy consumption for the prologue and for the epilogue, E_{PE} , is equal to $(s - 1) \times [\sum_{m \in O} N_m \times P_m + N_{IDLE_{PE}} \times P_{IDLE} + 2 \times II \times (P_{REST} + P_{CONF})]$ where O is a set of CGRA operations which is $\{ALU, MUL, DIV, ROUT, MEM\}$, N_m is the number of nodes for m operation (for example, N_{ALU} is the number of nodes for arithmetic and logic operation), and $N_{IDLE_{PE}}$ is the number of idle PEs in the prologue and the epilogue. The energy consumption for the kernel, E_K , is equal to $(i - s + 1) \times [\sum_{m \in O} N_m \times P_m + N_{IDLE_K} \times P_{IDLE} + II \times (P_{REST} + P_{CONF})]$ where N_{IDLE_K} is the number of nodes for idle nodes in the kernel. Thus, the total energy consumption E is represented as $E_K + E_{PE}$.

4.4.2 Experimental Results

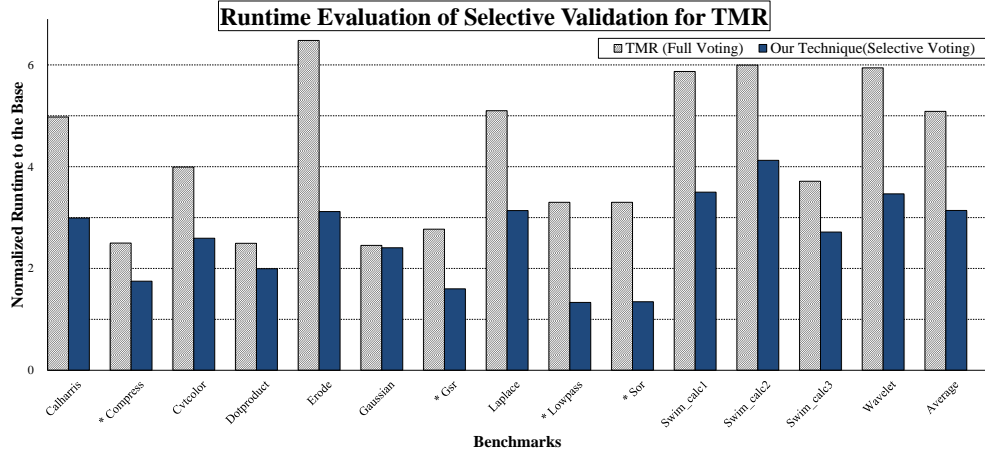
Effectiveness of Selective Validations

Our first set of experiments is to evaluate the effectiveness of our selective validations for software-implemented redundancy techniques on CGRAs in terms of the runtime and the energy consumption. Figure 4.9 clearly shows the effectiveness of our selective validation for TMR on CGRAs. Y-axis in Figure 4.9(a) represents the normalized runtime of TMR with the full voting and that of TMR with the selective voting (our approach) to that of the base. Over the suite of benchmarks, our selective validation for TMR can improve the performance in terms of the runtime by 38.3% on average as compared to that of the full validation for TMR on CGRAs. Y-axis in Figure 4.9(b)

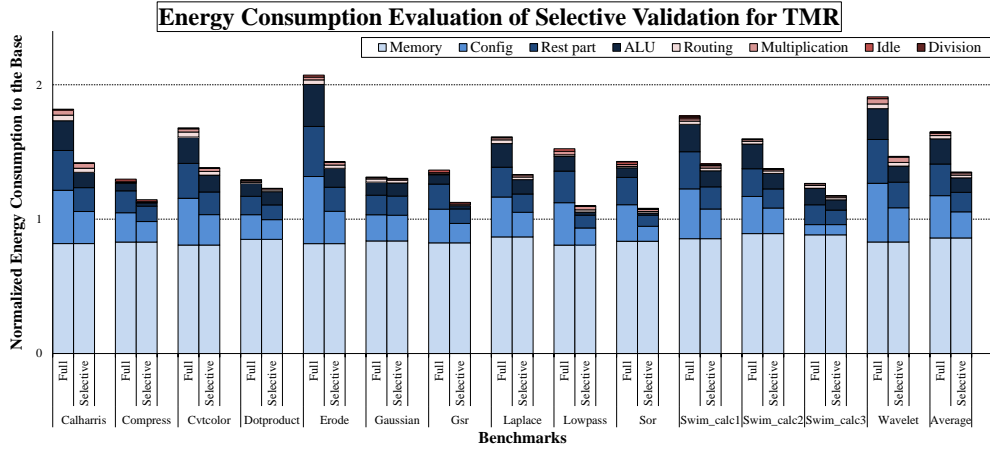
represents the normalized energy consumption of TMR with the full voting and that of TMR with the selective voting to that of the base. Over the suite of benchmarks, our selective validation for TMR can reduce the energy consumption by 18.1% on average as compared to that of the full validation for TMR on CGRAs. The main reason of these improvements of runtime and energy consumption is because our approach selects only synchronous operations, i.e., *store* operations, as validation points rather than every operation where the previously proposed TMR technique validates. Note that every voting requires additionally seven operations which are significantly expensive with respect to the runtime and the energy consumption on CGRAs. Figure 4.9(a) and Figure 4.9(b) show negligible improvements for benchmark *Gaussian* since it contains relatively small number of normal operations between memory ones. On the other hand, the other benchmarks contain the larger number of normal operations between memory ones where our approach can reduce the number of validations and improve the runtime and the energy consumption more effectively. Therefore, additional operations for triplication and voting can be covered by unused PEs for these benchmarks. In particular, *Lowpass* in TMR with the selective voting significantly improves the runtime (59.6%) than that in TMR with the full voting and *Erode* in TMR with the selective voting significantly improves the energy consumption (31.1%) than that in TMR with the full voting. Note that our approach triplicates every operation and can manage the comparable fault coverage as the previously proposed or conventional TMR technique does.

Interestingly, our selective voting techniques are more effective at reducing runtime for the benchmark kernels that include recurrent loops (benchmarks marked with the asterisk in Figure 4.9(a)). They can improve the runtime by 53.0% on average in the selective voting as compared to the full voting while the other benchmarks can improve the runtime by 37.8% on average. In the case of applying TMR with the full voting mechanism to these benchmarks, the critical path of recurrent data dependence, crucially affecting the RecMII, lengthens about three times more than the critical path without voting mechanism. Due to the bigger RecMII, MII, the maximum value of RecMII and ResMII, is also set to the value of RecMII that is much higher value than ResMII, so the II increases; i.e., the performance degrades due to the longer data dependence between iterations. In our approach, however, the RecMIIs of these benchmarks slightly increase since the critical path lengthens less than the full voting. Thus, our approach can achieve better performance than the TMR with the full voting in recurrent loop cases.

We also evaluate our software-implemented DMR with the selective comparison and DMR with the full comparison in terms of the runtime and the energy consumption. Figure 4.10 clearly shows that DMR with the selective comparison mechanism improves the runtime and the energy consumption. DMR with the full comparison duplicates and compares all the operations while our DMR with the selective comparison

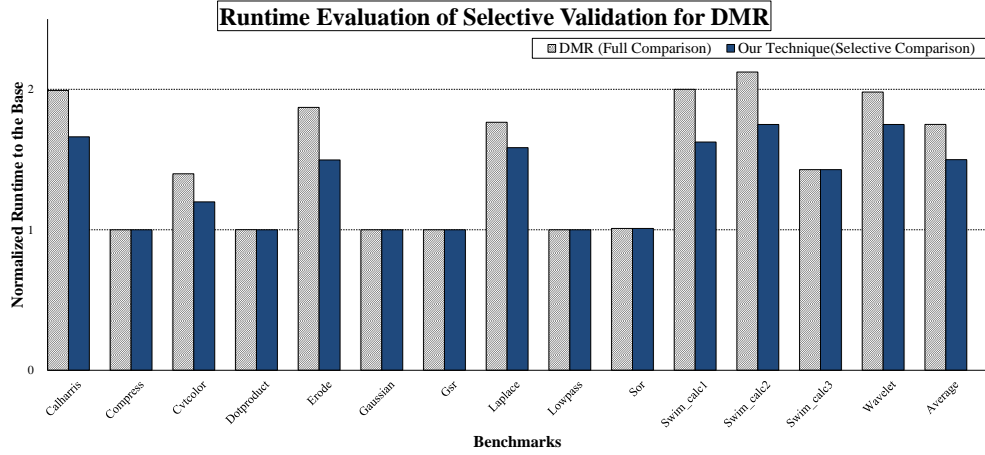


(a) Runtime evaluation of our selective voting (the asterisk indicates benchmarks with recurrent loops)

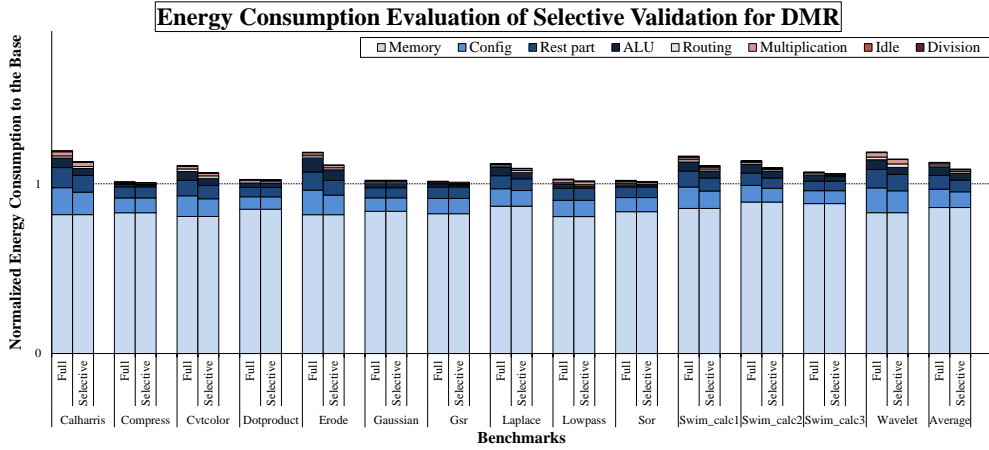


(b) Energy consumption evaluation of our selective voting

Figure 4.9 Our selective voting for TMR outperforms the full voting in terms of runtime and energy consumption.



(a) Runtime evaluation of our selective comparison



(b) Energy consumption evaluation of our selective comparison

Figure 4.10 Our selective comparison for DMR outperforms the full comparison in terms of runtime and energy consumption.

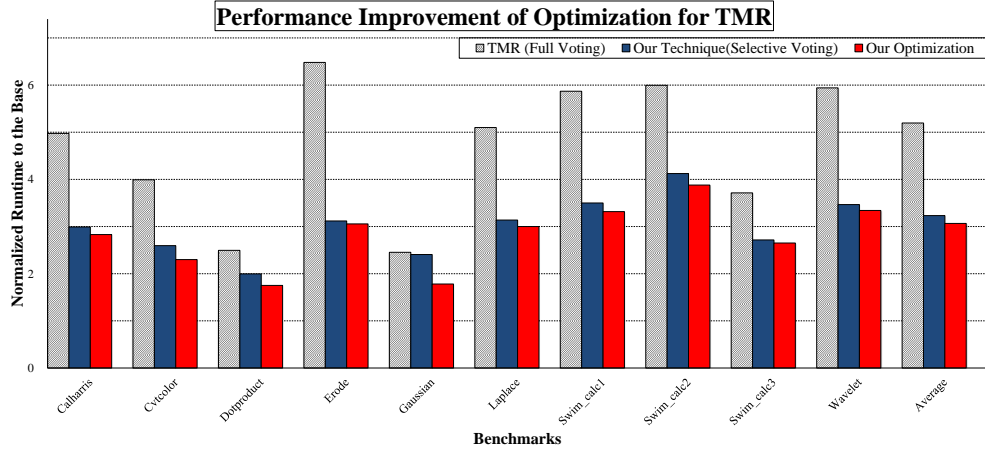
duplicates all the operations but compares only before a *store* operation is executed. We normalize the runtime of DMR with the full comparison and that of DMR with the selective comparison to that of the base as shown in Figure 4.10(a). Most benchmarks achieve runtime improvement (14.3% on average over benchmarks) with our selective comparison as compared to DMR with the full comparison. The benchmark *Erode* in DMR with the selective comparison achieves the maximum runtime improvement by 20.0%. And we normalize the energy consumption of DMR with the full comparison and that of DMR with the selective comparison to that of the base as shown in Figure 4.10(b). Most benchmarks achieve energy saving slightly (3.6% on average over benchmarks) as compared to DMR with the full comparison. The benchmark *Erode* in DMR with the selective comparison achieves the maximum energy consumption improvement by 6.4%. In general, DMR techniques with the selective validation achieve the less benefit in terms of the runtime and the energy consumption than TMR techniques mainly because DMR generates the smaller number of duplicated operations than triplicated operations in TMR. DMR needs additional two operations for implementing the comparison while TMR needs additional seven operations for implementing the voting. Thus, several benchmarks incur the runtime and the energy consumption overheads in the selective comparison for DMR close to those in the full comparison as shown in Figure 4.10.

Figure 4.9(b) and Figure 4.10(b) show the energy consumption distributions of local memory accesses (Memory), configuration memory accesses (Config), idle PE (Idle), active PEs of ALU (ALU), multiplication (Multiplication), division (Division), and routing (Routing), and the rest part of PE array (Rest part) as summarized in Table 4.1. There is no improvement with respect to energy consumption of local memory access operations from the full validation to the selective one due to no difference in the numbers of local memory accesses between them. The improvement of the selective validation as compared to the full validation is mainly influenced by operations of the configuration memory access, the rest part of PE array, and the active PE of ALU. The energy savings by operations of the configuration memory access, the rest part of PE array, and the active PE of ALU between the full validation and the selective validation are achieved by 9.8%, 7.3% and 6.6%, respectively, for TMR as shown in Figure 4.9(b) and 1.9%, 1.4% and 1.4%, respectively, for DMR as shown in Figure 4.10(b). The energy saving for the configuration memory access and the rest part of PE array mainly results from the decreased *II*. The DFG of the selective validation is simpler than that of the full validation so *II* of selective validations can decrease due to the reduced number of nodes and edges. Therefore, the energy saving in the active PE of ALU results from the reduced number of ALU nodes thanks to selective validation schemes. Other operations such as the idle PE and the active PE of multiplication, division and routing slightly reduce the energy consumption from the full validation to the selective one.

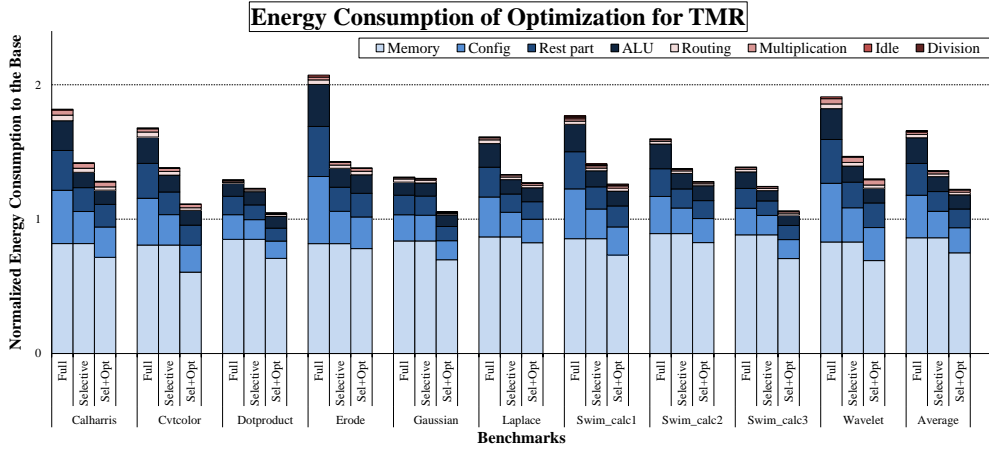
In summary, our selective validation techniques for TMR and DMR are significantly effective in terms of the runtime (by 38.3% and 14.3% on average) and the energy consumption (by 18.1% and 3.6% on average) as compared to the complete validation techniques for those redundancy techniques implemented in software on CGRAs.

Enhanced Effectiveness with Optimizations

Our second set of experiments is to evaluate our optimization technique by reducing the number of *store* operations where the validation mechanism needs to be applied. Figure 4.11 and Figure 4.12 clearly shows the effectiveness of our selective validation techniques with *store* operations reduced in terms of the runtime and the energy consumption. Note that benchmarks with the recurrent loops are excluded in this set of optimization experiments as explained in Section 4.4. Figure 4.11(a) shows that our optimization technique can improve the performance with respect to the runtime on average by 5.2% as compared to our own technique without the optimization and by 41.0% as compared to the previously proposed TMR technique with the full validation. Interestingly, our selective validation techniques with the optimization are effective in terms of the runtime for benchmark *Gaussian* (27.4% improvement) while it is less effective in the selective validation techniques without the optimization as shown in Figure 4.11(a). Figure 4.11(b) shows that our optimization technique can reduce the energy consumption by 10.1% on average as compared to our own technique without the optimization and by 26.2% as compared to the previously proposed TMR technique with the full validation. This energy saving for the optimization definitely results from the reduced number of local memory access operations as shown in Figure 4.11(b) and Figure 4.12(b). The energy savings by the local memory access operations from the full validation to our selective validation and optimization with *store* reduction are achieved by 6.8% for TMR and 9.9% for DMR. This effectiveness results from the reduced number of *store* operations obviously, and the power dissipation of the local memory access operation is relatively high as shown in Table 4.1. The energy savings by operations of the configuration memory access, the rest part of PE array, and the active PE of ALU between the full validation and the our optimization with *store* reduction are 7.8%, 5.8% and 5.2%, respectively, for TMR and 1.7%, 1.3% and 0.8%, respectively, for DMR. Figure 4.12(a) shows that our optimization technique for DMR by reducing the number of *store* operations can achieve the runtime on average by 17.8% as compared to the previously proposed DMR techniques with the full comparisons. Note that these optimization techniques by reducing the number of *store* operations show the high effectiveness on some benchmarks where there exist several *store* operations so that we can have enough margins to decrease the number of *store* operations, leading to the runtime improvement. On the contrary, benchmark

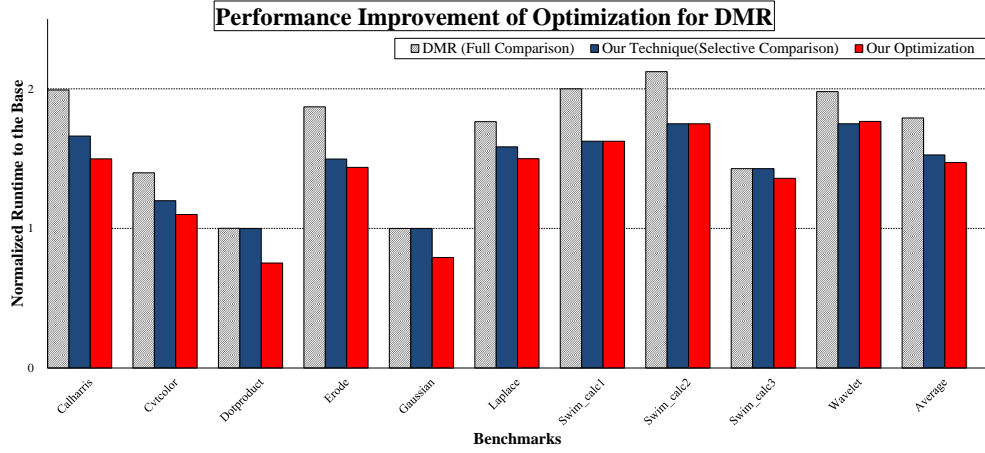


(a) Runtime improvement of our selective voting and optimization

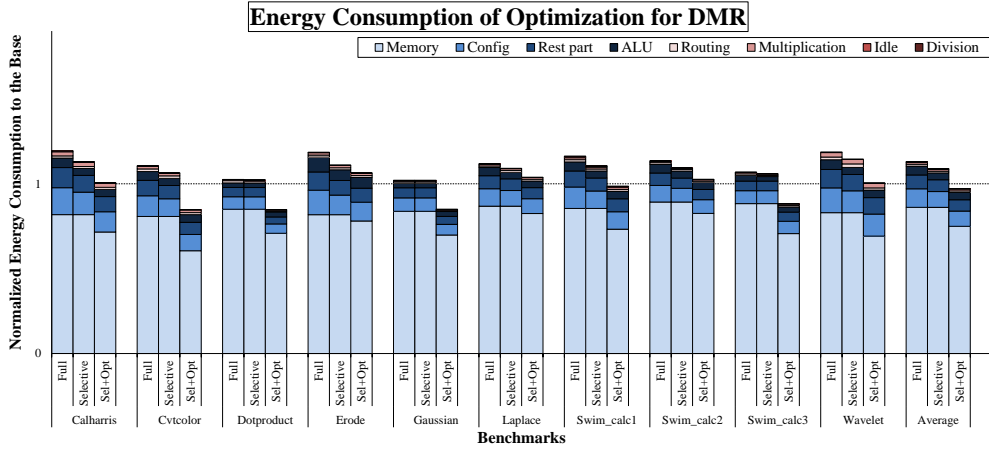


(b) Energy consumption of our selective voting and optimization

Figure 4.11 Our optimization techniques for TMR can improve the performance in terms of runtime and the energy consumption.



(a) Runtime improvement of our selective comparison and optimization



(b) Energy consumption of our selective comparison and optimization

Figure 4.12 Our optimization techniques for DMR can improve the performance in terms of runtime and the energy consumption.

Wavelet shows the runtime degradation since it has just two *store* operations where the *store* reduction rather incurs the runtime overhead due to the extra operations for merging these operations. Figure 4.12(b) shows that our optimization technique for DMR by reducing the number of *store* operations can achieve the energy consumption on average by 14.0% as compared to the previously proposed DMR techniques with the full comparisons. Interestingly, the energy consumption of our selective comparison with optimization technique for DMR is less than that of the base in benchmarks such as *Cvtcolor*, *Dotproduct*, *Gaussian*, *Swim_calc1*, and *Swim_calc3* as shown in Figure 4.12(b). Energy consumption is improved by 2.8% on average, and by up to 15.2% with benchmark *Cvtcolor*. In particular, both runtime and energy consumption of our optimization in benchmarks *Dotproduct* and *Gaussian* are less than that of the base as shown in Figure 4.12(a) and Figure 4.12(b). This effectiveness of optimization technique with selective DMR comparison can detect soft errors with less runtime and less energy consumption than the base in some benchmarks.

In summary, our optimization technique with the selective validation techniques for TMR and DMR can achieve the further improvement in terms of the runtime (by 41.0% and 17.8% on average) and the energy consumption (by 26.2% and 14.0% on average) as compared to the previously proposed redundancy techniques with the complete validation implemented in software on CGRAs.

Our last set of experiments is to show evaluations of the runtime and the energy consumption of the base and all redundancy techniques such as DMR with the full comparison, DMR with the selective comparison, DMR with the selective comparison and the *store* reduction, TMR with the full voting, TMR with the selective voting, and TMR with the selective voting and the *store* reduction over the benchmark, *Cvtcolor*. Clearly, TMR techniques demand higher overheads for the runtime and the energy consumption than the DMR ones as shown in Figure 4.13 while TMR ones are able to correct errors and DMR ones are not (they just detect them, i.e., they need the recovery mechanisms). Our proposals with the selective validation and the *store* reduction can achieve the better performance in terms of runtime and energy consumption than conventional DMR and TMR techniques implemented in software on CGRAs. Our optimization technique by reducing the number of *store* operations can incur the runtime overheads by 47.2% for DMR and 206.6% for TMR and energy consumption overheads by -2.8% for DMR and 22.2% for TMR on average over benchmarks as compared to the base. Note that previously proposed DMR and TMR techniques incur the runtime overheads by 79.1% and 419.6% and the energy consumption overheads by 13.0% and 65.7%, respectively, which are much higher than our selective validation techniques. Indeed, our selective techniques with the optimization can reduce the runtime overheads by 17.8% and 41.0% for DMR and TMR and also reduce the energy consumption overheads by 14.0% and 26.2% for DMR and TMR with the full validations, respectively. However, our selective validation techniques provide the compara-

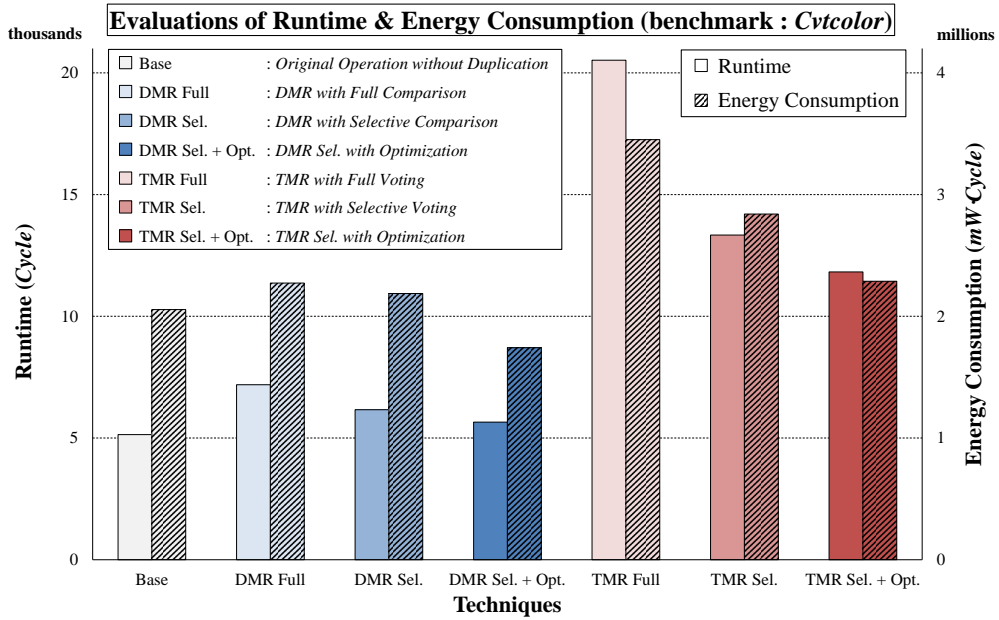


Figure 4.13 Evaluations of runtime and energy consumption among various protection techniques (benchmark: *Cvtcolor*)

ble reliability to conventional redundancy techniques. Note that these experiments can be expanded to guide designers or programmers to explore interesting tradeoff spaces between the runtime, the energy consumption, and the reliability, which is definitely our future work.

In summary, our evaluations of runtime and energy consumption show the efficacy of our selective techniques with the optimization and our various approaches for redundancy techniques can open a new venue for multidimensional tradeoff studies.

Chapter 5

Conculsion

In this dissertation, we attempt not only to reduce the instruction encoding bit width into 16-bit but also to suggest a strategy that extends the instruction bit width to 32 bits dynamically on demand with hardware and compiler support. In this attempt, we have introduced notions of partial instructions and remote operands which are not encoded in the instruction word but stored in a special slot called a ROA slot. Our compiler tries to have ROA slots occupy as many slots in the VLIW packets as possible that otherwise would be filled with NOPs, and synchronizes partial instructions with ROA slots. A partial instruction and its coupled remote operands combine together to form a complete instruction at run time. To prove the overall efficiency of our technique, we have conducted experiments with a set of benchmark programs. In our experiments, we obtain 28% energy reduction for the instruction fetch and 41% code size reduction on average in comparison with the 32-bit ISA VLIW processor, while the execution time slows down slightly. Further, we identify that our approach is well applied to a VLES (variable length execution set) architecture which is successfully adopted in modern VLIW processors. The experimental results, therefore, prove that the proposed strategy converts 32-bit ISA into an equivalent 16-bit ISA without loss of the original semantics, while leading to a substantial reduction of code size along with a potentially large reduction of power consumption and chip size at a slight run time cost.

Embedded systems designers are paying more and more attention to soft errors to increase reliability as well as other constraints such as power, performance, areas and code size. In VLIW, researchers proposed techniques to increase reliability by duplicating instructions. However, these techniques in general incur the increase of

code size and ignore different vulnerability levels of instructions. In this work, we propose compiler-assisted dynamic code duplication for VLIW architectures and present vulnerability-aware duplication algorithms to effectively increase reliability with minimal code size overhead. Future work includes the vulnerability analysis of instructions at other abstraction levels such as the gate level and further selective protections to achieve high reliability against soft errors with minimal overheads in terms of power, performance and area for other architectures.

Soft errors induced by radiation are receiving significant concerns since the soft error rate is increasing exponentially with aggressive technology scaling. CGRA with high performance and high flexibility becomes more and more popular even in critical applications such as finance programs, human health system, etc. In order to improve the reliability in CGRA, several fault tolerant techniques have been proposed but they incur area cost and performance degradation significantly. In order to protect the datapath in CGRAs from soft errors without area cost, we propose software-based selective validation techniques with the least performance overhead and the comparable fault coverage. We also propose an optimization technique by reducing the number of *store* operations to maximize the performance improvement. Our optimization technique merges multiple *store* operations into one *store* operation by DFG modification to reduce the number of validations. In conclusion, our selective validation techniques with the optimization can improve the runtime by 41.0% and the energy consumption by 26.2% as compared to the previously proposed TMR with the full validation. Future works include optimizing the operation of duplicating the original DFGs for applying redundancy techniques such as TMR and DMR with guaranteeing the comparable fault coverage and correct functionality in order to improve performance. We are also interested in investigating different priorities for various operations to apply the selective protection to only important or critical ones in terms of the reliability.

Bibliography

- [1] G. Lee and K. Choi, “Thermal-aware fault-tolerant system design with coarse-grained reconfigurable array architecture,” in *Adaptive Hardware and Systems (AHS), 2010 NASA/ESA Conference on*, 2010, pp. 265–272.
- [2] S. Segars, “Low power design techniques for microprocessors,” in *IEEE International Solid-State Circuits Conference Tutorial*, 2001.
- [3] *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide*, Texas Instruments Inc., Dallas, Texas, 2010.
- [4] S. Gurumurthi, A. Sivasubramaniam, M. J. Irwin, N. Vijaykrishnan, M. Kandemir, T. Li, and L. K. John, “Using Complete Machine Simulation for Software Power Estimation: The SoftWatt Approach,” in *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, 2002, pp. 141–.
- [5] A. Suga and K. Matsunami, “Introducing the FR500 Embedded Microprocessor,” *IEEE Micro*, vol. 20, pp. 21–27, July 2000.
- [6] *Xtensa Architecture White Paper*, Tensilica Inc, Santa Clara, CA, retrieved September, 2011, <http://www.tensilica.com/products/literature-docs/white-papers/xtensa-architecture/>.
- [7] A. Krishnaswamy and R. Gupta, “Profile guided selection of ARM and thumb instructions,” in *Proceedings of the joint conference on Languages, compilers and tools for embedded systems: software and compilers for embedded systems*, 2002, pp. 56–64.
- [8] A. Shrivastava and N. Dutt, “Energy efficient code generation exploiting reduced bit-width instruction set architectures (rISA),” in *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, 2004, pp. 475–477.

- [9] *MIPS32 Architecture for Programmers Volume IV-a: The MIPS16 Application Specific Extension to the MIPS32 Architecture*, MIPS Technology, Inc., Mountain View, CA, 2001.
- [10] D. Seal, *ARM Architecture Reference Manual(Second Edition)*. Addison-Wesley, 2001.
- [11] J. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. Irwin, "Compiler-assisted soft error detection under performance and energy constraints in embedded systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 8, no. 4, p. 27, 2009.
- [12] R. Baumann, "Soft errors in advanced computer systems," *Design & Test of Computers, IEEE*, vol. 22, no. 3, pp. 258–266, 2005.
- [13] P. Hazucha and C. Svensson, "Impact of cmos technology scaling on the atmospheric neutron soft error rate," *Nuclear Science, IEEE Transactions on*, vol. 47, no. 6, pp. 2586–2594, 2000.
- [14] F. Wrobel, J. Palau, M. Calvet, O. Bersillon, and H. Duarte, "Simulation of nucleon-induced nuclear reactions in a simplified sram structure: scaling effects on seu and mbu cross sections," *Nuclear Science, IEEE Transactions on*, vol. 48, no. 6, pp. 1946–1952, 2001.
- [15] A. Shrivastava, J. Lee, and R. Jeyapaul, "Cache vulnerability equations for protecting data in embedded processor caches from soft errors," in *ACM SIGPLAN Notices*, vol. 45, no. 4. ACM, 2010, pp. 143–152.
- [16] S. Mukherjee, *Architecture design for soft errors*. Morgan Kaufmann, 2008.
- [17] H. Zhong, K. Fan, S. Mahlke, and M. Schlansker, "A distributed control path architecture for vliw processors," in *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*. IEEE, 2005, pp. 197–206.
- [18] J. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. Irwin, "Compiler-directed instruction duplication for soft error detection," in *Design, Automation and Test in Europe, 2005. Proceedings*. IEEE, 2005, pp. 1056–1057.
- [19] C. Bolchini, "A software methodology for detecting hardware faults in vliw data paths," *Reliability, IEEE Transactions on*, vol. 52, no. 4, pp. 458–468, 2003.

- [20] J. Holm and P. Banerjee, "Low cost concurrent error detection in a vliw architecture using replicated instructions," in *Proceedings of the International Conference on Parallel Processing*, 1992, pp. 192–195.
- [21] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee, "Design and evaluation of hybrid fault-detection systems," in *Proceedings of the 32nd annual international symposium on Computer Architecture*, 2005, pp. 148–159.
- [22] Y. Kim, J. Lee, A. Shrivastava, and Y. Paek, "Operation and data mapping for CGRAs with multi-bank memory," *SIGPLAN Not.*, vol. 45, no. 4, pp. 17–26, Apr. 2010. [Online]. Available: <http://doi.acm.org/10.1145/1755951.1755892>
- [23] H. Singh, G. Lu, M.-H. Lee, E. Filho, R. Maestre, F. Kurdahi, and N. Bagherzadeh, "MorphoSys: case study of a reconfigurable computing system targeting multimedia applications," in *Design Automation Conference, 2000. Proceedings 2000*, 2000, pp. 573–578.
- [24] N. Wang, J. Quek, T. Rafacz, and S. Patel, "Characterizing the effects of transient faults on a high-performance processor pipeline," in *Dependable Systems and Networks, 2004 International Conference on*. IEEE, 2004, pp. 61–70.
- [25] D. Lyons, "Sun screen," 2000.
- [26] S. Michalak, K. Harris, N. Hengartner, B. Takala, and S. Wender, "Predicting the number of fatal soft errors in Los Alamos national laboratory's ASC Q supercomputer," *Device and Materials Reliability, IEEE Transactions on*, vol. 5, no. 3, pp. 329–335, 2005.
- [27] S. Jafri, S. Piestrak, O. Sentieys, and S. Pillement, "Design of a fault-tolerant coarse-grained reconfigurable architecture: a case study," in *Quality Electronic Design (ISQED), 2010 11th International Symposium on*, 2010, pp. 845–852.
- [28] D. Alnaji, Y. Ko, T. Imagawa, H. Konoura, M. Hiromoto, Y. Mitsuyama, M. Hashimoto, H. Ochi, and T. Onoye, "Coarse-grained dynamically reconfigurable architecture with flexible reliability," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, 2009, pp. 186–192.
- [29] T. Schweizer, A. Kuster, S. Eisenhardt, T. Kuhn, and W. Rosenstiel, "Using run-time reconfiguration to implement fault-tolerant coarse grained reconfigurable architectures," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, 2012, pp. 320–327.

- [30] T. Schweizer, P. Schlicker, S. Eisenhardt, T. Kuhn, and W. Rosenstiel, "Low-cost TMR for fault-tolerance on coarse-grained reconfigurable architectures," in *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, 2011, pp. 135–140.
- [31] C. Engelmann, H. Ong, and S. L. Scott, "The case for modular redundancy in large-scale high performance computing systems," in *Proceedings of the IASTED International Conference*, vol. 641, 2009, p. 046.
- [32] F. Kastensmidt, L. Sterpone, L. Carro, and M. Reorda, "On the optimal design of triple modular redundancy logic for SRAM-based FPGAs," in *Design, Automation and Test in Europe, 2005. Proceedings*, 2005, pp. 1290–1295 Vol. 2.
- [33] K. Singh, A. Agbaria, D. Kang, and M. French, "Tolerating SEU faults in the Raw architecture," in *International Workshop on Dependable Embedded Systems (WDES)*, 2006, p. 35.
- [34] S. Haga, A. Webber, Y. Zhang, N. Nguyen, and R. Barua, "Reducing code size in VLIW instruction scheduling," *J. Embedded Comput.*, vol. 1, pp. 415–433, August 2005.
- [35] J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Trans. Comput.*, vol. 30, pp. 478–490, July 1981.
- [36] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: an effective technique for VLIW and superscalar compilation," *J. Supercomput.*, vol. 7, pp. 229–248, May 1993.
- [37] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the 25th annual international symposium on Microarchitecture*, 1992, pp. 45–54.
- [38] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: maximizing on-chip parallelism," in *Proceedings of the 22nd annual international symposium on Computer architecture*, 1995, pp. 392–403.
- [39] *SC140 DSP Core Reference Manual*, Freescale Semiconductor, Inc., 2005.
- [40] *Design Compiler Reference Manual*, Synopsys Inc., Mountain View, CA, 2001.
- [41] M. Ahn and Y. Paek, "Fast Code Generation for Embedded Processors with Aliased Heterogeneous Registers," in *Transactions on High-Performance Embedded Architectures and Compilers II*, 2009, vol. 5470, pp. 149–172.

- [42] F. H. McMahon, "The Livermore Fortran kernels: A computer test of the numerical performance range," Lawrence Livermore National Laboratory, Tech. Rep. UCRL-53745, 1986.
- [43] V. uZivojnovic, J. Martinez, V. C. Schlager, and H. Meyr, "DSPSTONE: A DSP-oriented benchmarking methodology," in *Proceedings of the International Conference on Signal Processing Applications and Technology*, 1994.
- [44] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," HP Laboratories, Tech. Rep. HPL-2009-85, 2009.
- [45] S. Hines, J. Green, G. Tyson, and D. Whalley, "Improving Program Efficiency by Packing Instructions into Registers," in *Proceedings of the 32nd annual international symposium on Computer Architecture*, 2005, pp. 260–271.
- [46] D. Patterson and J. Hennessy, *Computer Organization and Design - The Hardware/Software Interface*. Morgan Kaufmann Publishers, 2005.
- [47] H. Lin and Y. Fei, "Harnessing horizontal parallelism and vertical instruction packing of programs to improve system overall efficiency," in *Proceedings of the conference on Design, automation and test in Europe*, 2008, pp. 758–763.
- [48] T. M. Conte, S. Banerjia, S. Y. Larin, K. N. Menezes, and S. W. Sathaye, "Instruction fetch mechanisms for vliw architectures with compressed encodings," in *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 29. Washington, DC, USA: IEEE Computer Society, 1996, pp. 201–211. [Online]. Available: <http://dl.acm.org/citation.cfm?id=243846.243886>
- [49] S. Jee and K. Palaniappan, "Performance evaluation for a compressed-vliw processor," in *Proceedings of the 2002 ACM symposium on Applied computing*, ser. SAC '02. New York, NY, USA: ACM, 2002, pp. 913–917. [Online]. Available: <http://doi.acm.org/10.1145/508791.508967>
- [50] C. Lefurgy, P. Bird, I.-C. Chen, and T. Mudge, "Improving code density using compression techniques," in *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 30. Washington, DC, USA: IEEE Computer Society, 1997, pp. 194–203. [Online]. Available: <http://dl.acm.org/citation.cfm?id=266800.266819>
- [51] Y. Xie, W. Wolf, and H. Lekatsas, "Code compression for embedded vliw processors using variable-to-fixed coding," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 14, no. 5, pp. 525–536, may 2006.

- [52] M. Collin and M. Brorsson, "Two-Level Dictionary Code Compression: A New Scheme to Improve Instruction Code Density of Embedded Applications," *Code Generation and Optimization, IEEE/ACM International Symposium on*, pp. 231–242, 2009.
- [53] *D950-CORE Specification*, SGS-Thomson Microelectronics, 1995.
- [54] G. C. S. de Araujo, "Code generation algorithms for digital signal processors," Ph.D. dissertation, Princeton University, 1997.
- [55] R. Sucher, R. Niggebaum, G. Fettweis, and A. Rom, "Carmel - A New High Performance DSP Core Using CLIW," in *Proceedings of the International Conference on Signal Processing Applications and Technology*, 1998, pp. 499–504.
- [56] *DSP Benchmark Results for the Latest VLIW-Based Processors*, Berkely Design Technology Inc., Oakland, CA, 2003.
- [57] M. Tremblay and Y. Tamir, "Support for fault tolerance in vlsi processors," in *Circuits and Systems, 1989., IEEE International Symposium on*. IEEE, 1989, pp. 388–392.
- [58] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. Kim, "Robust system design with built-in soft-error resilience," *Computer*, vol. 38, no. 2, pp. 43–52, 2005.
- [59] T. Austin, "DIVA: A reliable substrate for deep submicron microarchitecture design," in *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*. IEEE, 1999, pp. 196–207.
- [60] A. Meixner, M. Bauer, and D. Sorin, "Argus: Low-cost, comprehensive error detection in simple cores," in *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*. Ieee, 2007, pp. 210–222.
- [61] M. Gomaa and T. Vijaykumar, "Opportunistic transient-fault detection," in *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*. IEEE, 2005, pp. 172–183.
- [62] V. Reddy, E. Rotenberg, and S. Parthasarathy, "Understanding prediction-based partial redundant threading for low-overhead, high-coverage fault tolerance," in *ACM SIGOPS Operating Systems Review*, vol. 40, no. 5. ACM, 2006, pp. 83–94.
- [63] S. Reinhardt and S. Mukherjee, "Transient fault detection via simultaneous multithreading," *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2, pp. 25–36, 2000.

- [64] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August, “Swift: Software implemented fault tolerance,” in *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 2005, pp. 243–254.
- [65] G. Reis, J. Chang, N. Vachharajani, R. Rangan, D. August, and S. Mukherjee, “Software-controlled fault tolerance,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 2, no. 4, pp. 366–396, 2005.
- [66] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, “A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor,” in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*. IEEE, 2003, pp. 29–40.
- [67] S. Rehman, M. Shafique, F. Kriebel, and J. Henkel, “Reliable software for unreliable hardware: embedded code generation aiming at reliability,” in *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2011, pp. 237–246.
- [68] N. Nakka, K. Pattabiraman, and R. Iyer, “Processor-level selective replication,” in *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2007, pp. 544–553.
- [69] D. Borodin and B. H. H. B. Juurlink, “Instruction precomputation with memoization for fault detection,” in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2010, pp. 1665–1668.
- [70] N. Wang and S. Patel, “Restore: Symptom-based soft error detection in microprocessors,” *Dependable and Secure Computing, IEEE Transactions on*, vol. 3, no. 3, pp. 188–201, july-sept. 2006.
- [71] N. K. Soundararajan, A. Parashar, and A. Sivasubramaniam, “Mechanisms for bounding vulnerabilities of processor structures,” in *Proceedings of the 34th annual international symposium on Computer architecture*, ser. ISCA ’07. New York, NY, USA: ACM, 2007, pp. 506–515. [Online]. Available: <http://doi.acm.org/10.1145/1250662.1250725>
- [72] X. Vera, J. Abella, J. Carretero, and A. González, “Selective replication: A lightweight technique for soft errors,” *ACM Transactions on Computer Systems (TOCS)*, vol. 27, no. 4, p. 8, 2009.
- [73] J. Lee, J. Youn, J. Lee, M. Ahn, and Y. Paek, “Dynamic operands insertion for VLIW architecture with a reduced bit-width instruction set,” in *Parallel Dis-*

- tributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, may 2012, pp. 119–130.
- [74] S. Mukherjee, J. Emer, and S. Reinhardt, “The soft error problem: An architectural perspective,” in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*. IEEE, 2005, pp. 243–247.
 - [75] K. Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian, “Mitigating soft error failures for multimedia applications by selective data protection,” in *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*. ACM, 2006, pp. 411–420.
 - [76] G. Reis, J. Chang, and D. August, “Automatic instruction-level software-only recovery,” *Micro, IEEE*, vol. 27, no. 1, pp. 36–47, 2007.
 - [77] K. Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian, “Partially protected caches to reduce failures due to soft errors in multimedia applications,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 17, no. 9, pp. 1343–1347, 2009.
 - [78] G. Asadi, V. Sridharan, M. Tahoori, and D. Kaeli, “Balancing performance and reliability in the memory hierarchy,” in *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*. IEEE, 2005, pp. 269–279.
 - [79] S. Wang, J. Hu, and S. Ziaavras, “On the characterization of data cache vulnerability in high-performance embedded microprocessors,” in *Embedded Computer Systems: Architectures, Modeling and Simulation, 2006. IC-SAMOS 2006. International Conference on*. IEEE, 2006, pp. 14–20.
 - [80] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997, iISBN 1-5586-0320-4.
 - [81] S. Kim, “Area-efficient error protection for caches,” in *DATE’06*, Mar 2006, pp. 1282–1287.
 - [82] M. Ahn and Y. Paek, “Transactions on high-performance embedded architectures and compilers ii,” 2009, ch. Fast Code Generation for Embedded Processors with Aliased Heterogeneous Registers, pp. 149–172.
 - [83] V. Zivojnovic, J. Velarde, C. Schlager, and H. Meyr, “DSPstone: A DSP-oriented benchmarking methodology,” in *Proc. of the Intern. Conf. on Signal Processing and Technology*, 1994.

- [84] F. H. McMahon, "The Livermore Fortran kernels: A computer test of the numerical performance range," Lawrence Livermore National Laboratory, Tech. Rep. UCRL-53745, 1986.
- [85] S. Li, J. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*. IEEE, 2009, pp. 469–480.
- [86] S. Pillement, O. Sentieys, and R. David, "DART: a functional-level reconfigurable architecture for high energy efficiency," *EURASIP J. Embedded Syst.*, vol. 2008, pp. 5:1–5:13, Jan. 2008. [Online]. Available: <http://dx.doi.org/10.1155/2008/562326>
- [87] S. Eisenhardt, A. Kuster, T. Schweizer, T. Kuhn, and W. Rosenstiel, "Spatial and temporal data path remapping for fault-tolerant coarse-grained reconfigurable architectures," in *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2011 IEEE International Symposium on*, 2011, pp. 382–388.
- [88] Y. Kim and R. N. Mahapatra, "Dynamic context management for low power coarse-grained reconfigurable architecture," in *Proceedings of the 19th ACM Great Lakes symposium on VLSI*, ser. GLSVLSI '09. New York, NY, USA: ACM, 2009, pp. 33–38. [Online]. Available: <http://doi.acm.org/10.1145/1531542.1531555>
- [89] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The Raw microprocessor: a computational fabric for software circuits and general-purpose programs," *Micro, IEEE*, vol. 22, no. 2, pp. 25–35, 2002.
- [90] B. Mei, S. Vernalde, D. Verkest, and R. Lauwereins, "Design methodology for a tightly coupled VLIW/reconfigurable matrix architecture: A case study," in *Proceedings of the conference on Design, automation and test in Europe - Volume 2*, ser. DATE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 21 224–. [Online]. Available: <http://dl.acm.org/citation.cfm?id=968879.969178>
- [91] J. Chang, G. Reis, and D. August, "Automatic instruction-level software-only recovery," in *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, 2006, pp. 83–92.
- [92] B. R. Rau, "Iterative modulo scheduling: an algorithm for software pipelining loops," in *Proceedings of the 27th annual international symposium on*

- Microarchitecture*, ser. MICRO 27. New York, NY, USA: ACM, 1994, pp. 63–74. [Online]. Available: <http://doi.acm.org/10.1145/192724.192731>
- [93] K. Lee, A. Shrivastava, M. Kim, N. Dutt, and N. Venkatasubramanian, “Mitigating the impact of hardware defects on multimedia applications: a cross-layer approach,” in *Proceedings of the 16th ACM international conference on Multimedia*, ser. MM ’08. New York, NY, USA: ACM, 2008, pp. 319–328. [Online]. Available: <http://doi.acm.org/10.1145/1459359.1459402>
 - [94] Y. Kim, J. Lee, A. Shrivastava, and Y. Paek, “Memory access optimization in compilation for coarse-grained reconfigurable architectures,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 16, no. 4, pp. 42:1–42:27, Oct. 2011. [Online]. Available: <http://doi.acm.org/10.1145/2003695.2003702>
 - [95] G. Bradski, “The OpenCV library,” *Doctor Dobbs Journal*, 2000.
 - [96] J. L. Henning, “SPEC CPU2000: measuring cpu performance in the new millennium,” *Computer*, vol. 33, no. 7, pp. 28–35, 2000.
 - [97] Y. Kim, J. Lee, T. X. Mai, and Y. Paek, “Improving performance of nested loops on reconfigurable array processors,” *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 32:1–32:23, Jan. 2012. [Online]. Available: <http://doi.acm.org/10.1145/2086696.2086711>

초록

오늘날 내장형 프로세서 설계는 저전력, 저면적, 고성능을 동시에 만족시켜야 하는 제약을 갖는다. 이러한 요건들의 충족을 저해하는 요인 중 대표적인 것으로 명령어 메모리와 캐쉬로부터 명령어를 읽어들이는 과정을 들 수 있다. 명령어 메모리와 캐쉬의 크기는 전체 칩 면적에 큰 비중을 차지하고, 또한 캐쉬로의 잦은 접근은 높은 전력 소모를 유발하며, 캐쉬 미스로 인한 전체 시스템의 성능 저하를 가져온다. 따라서, 이러한 명령어 전달 과정의 부정적인 영향을 줄이기 위해서, 본 논문에서는 코드 크기 최적화를 통한 명령어 메모리와 캐쉬의 크기 결정에 초점을 맞추고자 한다.

본 논문에서는 코드 크기 최적화 적용이 가능한 세 가지 현상에 주목하였다. 첫째로 VLIW 아키텍처에서 긴 명령어 비트폭 때문에 필요 이상의 전력과 메모리 공간이 소모된다는 점이다. 이의 해결 방안으로 감소된 비트폭을 가진 명령어 집합 아키텍처를 채택하는 것을 생각해볼 수 있다. 하지만 실제로 주어진 임의의 명령어 집합을 그와 동일한 동작을 유지한 채 비트폭을 감소시키는 것은 불가능에 가까운데, 이는 명령어 비트폭 감소로 인해 기존 명령어 집합의 일부분에 해당하는 정보만을 인코딩할 수 있기 때문이다. 본 논문에서는 기존의 32비트 길이의 명령어 집합을 갖는 프로세서를 정보 손실 없이 16비트 길이의 명령어 집합을 갖는 프로세서로 변형시키기 위해, 동적 암시 어드레싱 모드(DIAM)를 사용하여 원래의 비트폭의 절반의 비트폭을 가지면서 동일하게 동작하는 VLIW 아키텍처를 제안한다.

둘째로 소프트 에러에 대항하여 신뢰도를 높이기 위해 VLIW 와 같이 동시에 여러 개의 명령어들을 실행할 수 있는 구조의 내장형 시스템에서 코드 복제 기법이 사용되고 있다는 점이다. 복제된 코드는 VLIW 의 빈 슬롯을 최대한 활용하여 오버헤드를 극복하고자 하지만, 복제 코드를 위한 추가적인 VLIW 패킷의 생성을 모두 막을 수는 없다. 이로 인해 신뢰도를 높일 수는 있어도 코드 크기 증가는 불가피해진다. 이러한 코드 크기 증가를 막고자 본 논문에서는 컴파일러의 보조를 통한 동적 코드 복제 기법을 제안한다. 이 기법을 사용하는 VLIW 아키텍처는 복제 코드를 생성하기 이전의 본래 코드와 같은 크기의 코드를 입력으로 받는데, 이 코드를 이루는 각각의 명령어에는 컴파일 시간에 생성된 정보가 내재되어 있다. 이 내재된 정보는 실행 시간에 동적으로 복제 코드를 생성할 수 있도록 해준다. 이와 같이 더 이상 복제 명령어가 코드에 존재하지 않기 때문에 제안된 기법에서는 복제 명령어들로 인한 코드 크기 증가를 피할 수 있다.

마지막 셋째는 두번째 경우와 마찬가지로 소프트 에러에 대항하기 위한 목적으로 CGRA 상에 삼중 모듈 중복 (TMR) 기법을 소프트웨어적으로 구현한 기법이 사용되고 있는 점이다. 이러한 방식은 CGRA 의 코드 크기를 증가시키는데, 이는 중복된 명령어들의 수행 결과를 검증하기 위해 사용되는 투표 기법에 많은 수의 명령어들이 필요하기 때문이다. 결국 CGRA 에서 수행해야 할 명령어들이 증가한 셈이기 때문에 결과적으로 실행 시간과 전력 소모의 증가를 야기시킨다. 본 논문

서는 이러한 성능 저하를 최소화하기 위한 목적으로 모든 명령어에 대해서 타당성 검증하는 것이 아니라 선택적 투표 기법으로 타당성 검증을 하는 방식을 제안한다.

주요어: 내장형 프로세서, 코드 크기, VLIW 아키텍처, 감소된 비트폭을 가진 명령어 집합 아키텍처, DIAM, 소프트 에러, 명령어 복제, CGRA, TMR, 선택적인 타당성 검증
학번: 2007-21051